Universidad Carlos III de Madrid

Departamento de Ingeniería de Sistemas y Automática

Proyecto Fin de Carrera

Reconocimiento de textos antiguos mediante técnicas de visión artificial

Autor: Moisés Rodríguez Vallejo
Tutor: Jorge García Bueno

Leganés, junio de 2013

Índice general

Índice de figuras

Índice de cuadros

Resumen

El objetivo de este proyecto es reconocer caracteres de textos antiguos mediante técnicas de visión artificial y redes neuronales. Distinguimos dos fases en el proceso: extracción de los caracteres de las imágenes de los textos antiguos; y el reconocimiento mediante redes neuronales. En la primera utilizaremos las librerías de OpenCV para conseguir el objetivo, mientras que en la segunda fase partimos de un código de una red neuronal de 3 capas de la Universidad Carnegie Mellon, modificado adecuadamente a nuestro propósito. También estudiaremos la influencia en nuestros resultados de varios parámetros de la red neuronal.

Abstract

The aim of this project is to recognize characters in antique books, using computer vision and neural networks techniques. This process has two phases: extracting characters from images of antique books; and the recognition using neural networks. In the first phase we will work with OpenCV libraries to reach the goal, while in the second we start from a 3-layer neural network code from the Carnegie Mellon University, adapted to our purpose. We will also analyze the influence of several paramethers on our neural network results.

Capítulo 1

Introducción y objetivos

1.1. Introducción

En esta introducción situaremos nuestro proyecto en su contexto. En primer lugar hablaremos de los textos antiguos con los que trabajamos, ya que una de las motivaciones de este proyecto es el uso de la ingeniería en un ámbito humanístico como son los textos góticos. Además, hablaremos sobre la digitalización de libros, principal tema de este proyecto. Nombraremos también un par de técnicas novedosas relacionadas con nuestro trabajo: por un lado un sistema automatizado de escaneo de libros, y por otro reCAPTCHA, un sistema que, indirectamente, ayuda a resolver problemas relacionados con el reconocimiento de caracteres.

1.1.1. Textos antiguos

En primer lugar introduciremos a los principales protagonistas de este proyecto: los textos antiguos. Hemos trabajado con textos góticos (siglos XII-XV d.C.), cuyas características se describirán a continuación.

Introducción a la escritura escritura gótica

Durante los siglos XII y XIII se producen en la cultura cambios radicales que repercuten sobre las características de los manuscritos y sobre la forma de escritura. Hasta esa época, los estudios y los libros eran un privilegio casi exclusivo de la Iglesia y únicamente producidos en monasterios y escuelas catedralicias. A partir de la mitad del siglo XII la cultura se difunde fuera de los monasterios, divulgándose en torno a las grandes universidades que van surgiendo en toda Europa y a las que acuden estudiosos de toda condición: eclesiásticos, religiosos y laicos. Por tanto surge la necesidad de proveer de libros a las universidades para poder desarrollar sus enseñanzas. Aparece

una gran demanda de textos que afecta al mercado, surgiendo un comercio en torno a los libros. Debido a estas nuevas necesidades, los manuscritos adoptan características distintas, adaptándose a la producción en serie. También se busca la economía de papel, es decir, que quepa la máxima cantidad de texto en el menor espacio posible, lo que implica contraer las letras, trazar ascendentes y descendentes cortos, así como utilizar abundantes abreviaturas. Otro factor que influyó en el aspecto de la letra gótica fue uno de tipo técnico: la punta de la pluma se cortó de forma oblicua a la izquierda, lo que da como resultado que los trazos horizontales y verticales fueran de trazo grueso y los oblicuos finos y tenues. Este detalle dificulta el algoritmo de segmentación de caracteres utilizado en este proyecto, como veremos más adelante.
Todas las formas de escritura que presentan las características mencionadas anteriormente son designadas con el nombre genérico de escritura *gótica*. Esta denominación procede de los humanistas italianos y es usada en oposición a la escritura "*antigua*" (*antiqua*) que indicaba la minúscula carolina, de la que deriva. En su época la escritura gótica recibía el nombre de "*moderna*" (*litterae modernae*).

Los primeros ejemplos de esta escritura (*littera protogothica / littera praegothica*) se encuentran en algunos manuscritos del siglo XII producidos en Francia, pero es en el siglo XIII cuando alcanza su perfección y se difunde rápidamente por toda Europa. La gótica fue usada en todo el mundo latino hasta el siglo XV, si bien su uso siguió vigente en Alemania hasta el siglo XVI por razones religiosas, no desapareciendo de hecho en este país en su variante "fraktur" hasta el siglo XX en que fue prohibida su utilización por un decreto de Hitler promulgado en enero de 1941.

Durante la evolución de la escritura gótica aparecieron algunos cambios en el alfabeto, tales como distintos trazos, distinciones entre letras que se confundían habitualmente (origen del punto sobre la "i", diferencia entre "u" y "v"), distintas representaciones para una misma letra según su posición en la palabra, inclusión de letras anteriormente extrañas("w", "y", "z"), etc. A finales del siglo XV influencias de la escritura humanística provocaron nuevos cambios en las letras góticas. Finalmente debe tenerse en cuenta que en la época clásica no había distinción en la escritura entre letras mayúsculas y minúsculas. Fue durante la época gótica cuando se estableció un auténtico alfabeto dual, es decir, entró en vigor el uso consistente de mayúsculas y minúsculas.

Podemos hacer una primera clasificación de tipos de escritura gótica en tres:

- Formal (*littera textualis / textura)*

- Cursiva (*littera gothica cursiva*)
- Bastarda o híbrida (*littera bastarda / littera hybrida)*

A continuación hablaremos brevemente sobre cada una de ellas.

Escritura gótica formal o textura

La variante textualis o textura es la más formal y cuidada de todos los tipos de letra gótica y fue usada en manuscritos de lujo, siendo hoy en día la forma más asociada con la escritura gótica. Toma su nombre del latín "textum" que significa entrelazado, tejido. Las distintas letras que forman una palabra están muy juntas, con tendencia incluso a solaparse algunos trazos, como si estuvieran "tejidas", siendo fácil confundir ciertas letras. Aparece un gran número de abreviaturas que ahorran espacio y tiempo al escribir, motivadas por las necesidades de producción del mercado.

La gótica textura presenta diversas variantes que se pueden agrupar en una jerarquía de cuatro grados, desde la escritura más desarrollada (y de mayor calidad) a la menos desarrollada:

- *prescissa*
- *quadrata*
- *semiquadrata*
- *rotunda*

A partir del siglo XII unas variantes más pequeñas y simplificadas de la gótica textura fueron introducidas para glosas o comentarios que acompañaban el texto de muchas obras, así como para libros de pequeño formato como las biblias en miniatura del siglo XIII. Estas escrituras eran más rápidas de escribir, muy compactas y empleaban cualidades cursivas. La variante usada para glosas es conocida como *littera gothica glossularis*, y para comentarios *littera gothica notularis* o *littera notula.*

Escritura gótica cursiva

Constituye la escritura de uso común en todo tipo de documentos, pero también fue utilizada como escritura libraria especialmente para textos escritos en lengua vulgar. Se desarrolló a finales del siglo XIII como forma simplificada de la textura, ya que se necesitaba una letra legible de escritura más rápida. Las letras se trazan de manera continua y las abreviaturas son muy frecuentes, para favorecer la velocidad de escritura.

Esta rapidez a la hora de escribir se aprecia en la forma de los trazos. Su uso se extendió a gran número de escribas (por el aumento de la demanda de libros) que introdujeron múltiples variaciones, por lo que existen muchos estilos al no haber unos estándares definidos.

Escritura gótica bastarda

Como su propio nombre indica, está constituida por elementos tanto de la escritura textura como de la cursiva. Aunque parte de una base informal, podemos encontrar escritura bastarda en manuscritos de lujo, acompañada de miniaturas muy decoradas, consiguiendo rapidez y fluidez al mismo tiempo que un aspecto cuidado.

Escritura gótica en España

Las órdenes monásticas pierden la hegemonía que habían ejercido durante siglos en el campo librario, ya que de las nuevas órdenes (franciscanos, dominicos, cartujos ...) ninguna se distingue por su dedicación especial a la producción de códices y a la conservación de los mismos. La secularización de la cultura y la aparición de las Universidades hacen que la balanza del asunto librario se incline a favor de las instituciones y personas seglares. La principal diferencia con respecto a Europa es la nomenclatura utilizada para etiquetar las escrituras góticas cursivas hispanas (documentarias: letra de privilegios, albalaes, precortesana, cortesana y procesal, según el uso que se les dio). En España hay fundamentalmente 3 tipos de escrituras librarias:

- Gótica caligráfica, fracturada o perfecta; coincide en época con la rotunda; se emplea para los manuscritos más solemnes o de lujo.
- Gótica redonda o semigótica: menos caligráfica.
- Gótica cursiva: con un trazado bastante irregular.

En cuanto a las escrituras góticas documentales hay un hecho de gran importancia como es la sustitución de la lengua latina en los documentos públicos y privados por las lenguas y dialectos romances. Por otra parte, la producción documental crece enormemente debido a la aparición y desarrollo de un fuerte y bien preparado cuerpo de notarios o escribanos públicos profesionales. Como consecuencia, la cursiva se adueña por completo del campo documentario, quedando reservado el uso de los tipos más formales para los documentos muy solemnes. Una característica de la escritura cursiva es la disminución del número de las abreviaturas en los documentos redactados en lenguas romances, pues cuanto más cursiva es una escritura, menos abreviaturas necesita.

La escritura castellana gótica no difiere en lo esencial de los rasgos anteriormente resumidos para las escrituras góticas del resto de Europa. Existe también, en el caso castellano, una progresiva cursividad, cuyo máximo exponente es la escritura cortesana. Sí existe en cambio un ligero desfase cronológico, ya que el resto de Europa optó por la sencillez gráfica mientras en España se usaba aún la escritura cortesana.

A continuación mostramos algunos de los textos empleados en este proyecto:

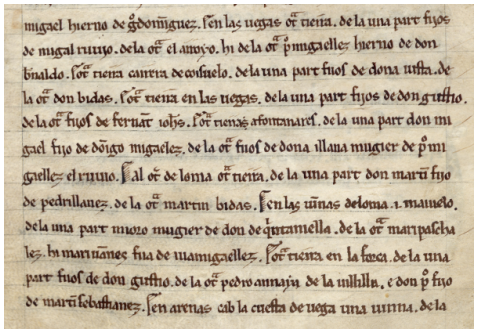

Figura 1.1: Ejemplo de texto antiguo 1

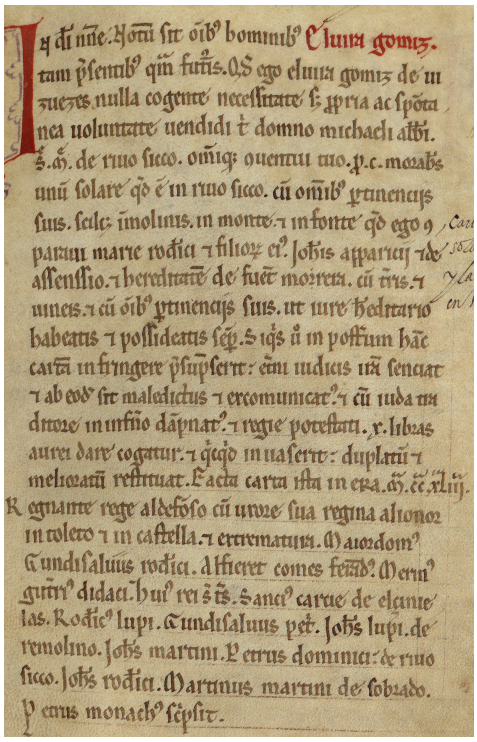

Figura 1.2: Ejemplo de texto antiguo 2

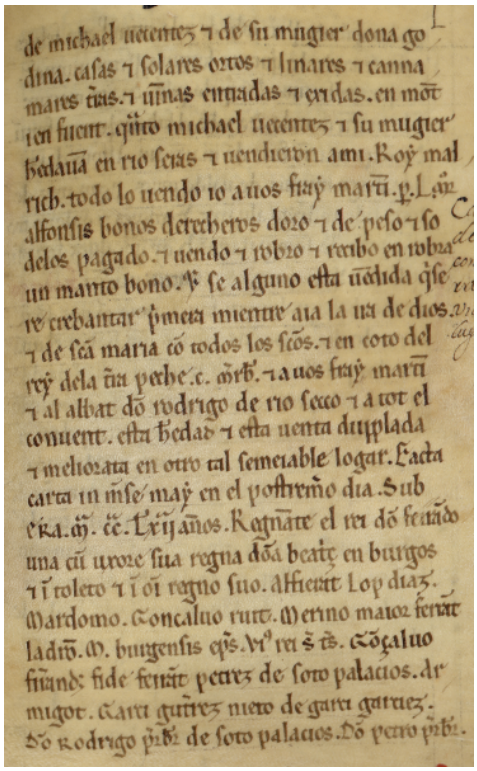

Figura 1.3: Ejemplo de texto antiguo 3

1.1.2. Digitalización de libros

Actualmente se tiende a minimizar el uso de papel en multitud de aplicaciones cotidianas. Podemos realizar un alto porcentaje de gestiones del día a día a través de nuestro ordenador con conexión a internet, ya que todos los documentos que pudiéramos necesitar para ello están digitalizados. Esta tendencia ha llegado también a los libros. Podemos tener toda una colección de títulos en un libro electrónico o tableta que ocupa menos espacio que un sólo libro tradicional. Por este motivo son muchos los esfuerzos dedicados a la digitalización de información en formato de papel, ya que podemos almacenar de manera ordenada infinidad de estantes de una biblioteca en un solo pendrive.

Este proyecto nace con esa finalidad: desarrollar un algoritmo capaz de digitalizar el texto de fotografías realizadas a las páginas de un libro. Lógicamente es una propuesta más modesta que algunas de las que mostraremos a continuación, pero el objetivo que las mueve es el mismo. En primer lugar hablaremos del reconocimiento óptico de caracteres (OCR en inglés), después de nuevos métodos para digitalizar libros y finalmente de optimizar y mejorar la digitalización de textos gracias a la colaboración de usuarios de internet.

Reconocimiento Óptico de Caracteres

Como ya hemos adelantado en el apartado anterior, el objetivo de este proyecto es digitalizar textos escritos. Para ello, dentro de la disciplina de la visión por computador, encontramos el reconocimiento óptico de caracteres, en el que por medio de una cámara o dispositivo óptico adquirimos una imagen, la tratamos adecuadamente y obtenemos información acerca de los caracteres que aparecen en ella.
Digitalizar textos con tipografía de imprenta es relativamente sencillo, ya que un mismo caracter aparece de manera casi idéntica a lo largo de todo el texto y los espacios y distancias existentes entre caracteres se pueden considerar constantes. En cambio, el texto manuscrito conlleva más dificultades, ya que al tratarse de un trazo continuo y menos uniforme, aparecen errores a la hora de segmentar los distintos caracteres y por tanto su reconocimiento es más complicado.
En general, el reconocimiento óptico de caracteres comienza con una etapa de acondicionamiento de la imagen, en la que intentamos eliminar posible ruido existente que pueda inducir posibles errores. Una vez filtrada la imagen, procedemos al binarizado de la misma, obteniendo una imagen únicamente en blanco y negro. El siguiente paso es la segmentación de la imagen, es decir, encontrar todos los posibles caracteres presentes en ella. Una vez segmentada, adelgazamos las componentes de los caracteres para permitir la posterior fase de comparación con patrones. En nuestro caso, no empleamos comparación con patrones, sino que utilizamos una red neuronal que "aprende" a reconocer cada uno de los caracteres. Se explicará en detalle en el apartado de implementación.
Existen multitud de aplicaciones comerciales de ROC, debido a su gran utilidad, aunque existen aún algunas limitaciones (por ejemplo la del texto manuscrito comentada anteriormente). A continuación hablaremos de una de las técnicas más novedosas de digitalización de libros.

BFS: Book Flipping Scanning

Ya hemos hablado sobre el reconocimiento óptico de caracteres tradicional y sus limitaciones. Pero estos métodos son lentos y, en ocasiones, poco eficaces. Es necesario escanear página a página o realizar fotos y supone un trabajo laborioso. A continuación hablaremos sobre una de los métodos más novedosos que podemos encontrar en la actualidad.
La Universidad de Tokyo ha desarrollado un sistema capaz de digitalizar hasta 250 páginas por minuto automáticamente, sin ningún tipo de operación humana. Este sistema, denominado BFS-Auto (Book Flipping Scanning), tiene tres aspectos clave: un

mecanismo completamente automatizado para pasar de página a alta velocidad; reconocimiento 3D en tiempo real de las páginas pasadas; y una conversión de alta precisión a una imagen de texto. En este caso no se reconoce cada uno de los caracteres, pero podría considerarse un primer paso para posteriormente utilizar un software OCR para obtener un fichero de texto. Con este sistema podemos escanear un libro de 250 páginas en un solo minuto sin ningún tipo de esfuerzo. Las cámaras de alta velocidad toman hasta 500 imágenes por segundo, reconociendo el mejor momento para digitalizar el texto. Además, consigue transformar la página deformada (por el paso de página) en una página completamente plana. Se muestran algunas imágenes a continuación.

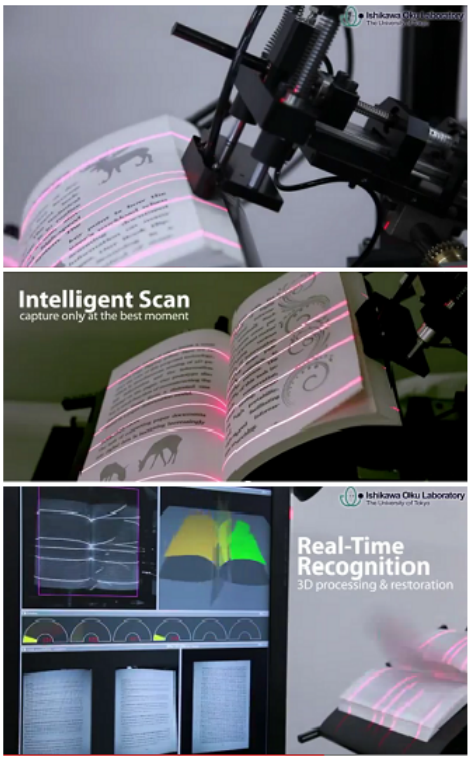

Figura 1.4: BFS-Auto

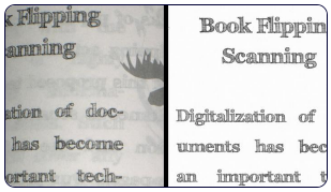

Figura 1.5: Resultado de la transformación de la imagen

reCAPTCHA

Relacionado con el reconocimiento de caracteres, la seguridad en internet y las redes neuronales encontramos reCAPTCHA, un servicio gratuito disponible en la red. Proviene de CAPTCHA (Completely Automated Public Turing Test To Tell Computers and Humans Apart), un sistema que nos indica si un usuario es humano o una máquina, para prevenir ataques en internet.
Pero en este caso, se aprovecha de una manera brillante la colaboración de los usuarios. Con CAPTCHA, al intentar acceder a multitud de webs, nos piden introducir los caracteres que aparecen en una imagen ligeramente distorsionada para garantizar que no somos una máquina. Pues bien, con reCAPTCHA realizan la misma labor y además ayudan a digitalizar libros. En este caso emplean dos palabras, la primera es una palabra de la cual conocen perfectamente su reconocimiento, y la segunda es una palabra encontrada en un libro que no consiguen reconocer mediante OCR.

Figura 1.6: Sistema reCAPTCHA

Si el usuario escribe correctamente la primera palabra, se considera que su respuesta a la segunda es también correcta, ayudando al OCR que no pudo reconocer dicha palabra. Obviamente no se sirve de una única persona, sino que el sistema prueba esa palabra un gran número de veces, garantizando un resultado correcto. Se puede encontrar una cierta similitud con un sistema de aprendizaje supervisado (el cual veremos en apartados posteriores), pero empleando a los usuarios de la aplicación para indicar los valores verdaderos para cada ejemplo. Según sus creadores, se resuelven unos 200 millones de CAPTCHA al día, lo que supone unas 150.000 horas de trabajo diarias que pueden emplearse para digitalizar libros.

1.2. Objetivos

El objetivo de este proyecto es crear un código capaz de reconocer caracteres de textos antiguos mediante redes neuronales. Obviamente no resulta tarea fácil, ya que estos textos presentan muchos inconvenientes de los que hablaremos en sus respectivos apartados, pero se obtienen buenos resultados en muchos casos. Para ello, se han realizado diversas pruebas modificando parámetros básicos de la red neuronal y comprobando su influencia en los resultados obtenidos, en términos de tiempo de convergencia de nuestra red y error cometido. Lógicamente no es un trabajo definitivo, pero es una buena primera aproximación para resolver este problema de reconocimiento de texto manuscrito.

1.3. Medios empleados

En primer lugar ha sido necesario digitalizar los textos antiguos. Para ello, se ha empleado un escáner *I2S Digibook Suprascan*, que nos permite obtener imágenes de 5358x4761 píxeles. De esta manera, tenemos un gran nivel de detalle, adecuado a nuestro propósito.
Se trata de un proyecto de programación, por lo que el dispositivo y sistema operativo empleados son importantes. En nuestro caso, hemos trabajado en *Ubuntu* (*12.10 Quantal Quetzal*) a través de una máquina virtual (*Oracle VM VirtualBox*, versión 4.1.18) en un procesador *Intel(R) Core(TM) i3 550 @ 3.20 GHz* con 1GB de memoria RAM y una tarjeta gráfica *NVIDIA GeForce 315*.
En cuanto al software empleado, hemos trabajado con *gedit* para editar el código y con el compilador *gcc*. Como herramienta de programación de aplicaciones de visión por computador, utilizamos las librerías de *OpenCV 2.4.4.0*.

1.4. Estructura de la memoria

Para facilitar la lectura de este documento, expondremos un esquema a continuación explicando brevemente cada uno de los apartados.

- Introducción: se pondrá en situación al lector sobre el objetivo de este proyecto, las disciplinas que podemos encontrar en él, el trabajo realizado y los medios empleados.

- Estado del arte: lugar para explicar la base teórica de la que consta este proyecto. En nuestro caso está dividida en dos apartados: por un lado la visión por

computador y por otro las redes neuronales.

- Implementación: en este capítulo encontramos la explicación detallada del algoritmo desarrollado.
- Experimentación y resultados: se analizan las pruebas realizadas y se exponen los resultados obtenidos.
- Conclusiones y trabajos futuros: capítulo en el que hacemos balance de todo el trabajo realizado y proponemos nuevas vías de trabajo.
- Anexos y bibliografía

Capítulo 2

Estado del arte

En este capítulo hablaremos de los conceptos principales relativos a este proyecto. En primer lugar trataremos conceptos relacionados con la visión por computador, principal disciplina contenida en este proyecto. A continuación, explicaremos la base teórica necesaria para comprender qué es lo que hace nuestra red neuronal en este trabajo. Se profundiza en los conceptos realmente aplicados en el algoritmo diseñado, aunque es necesario explicar mínimamente otros para poder dar consistencia a la redacción.

2.1. Imágenes digitales

El principal objeto de trabajo en este proyecto son los textos, que manipulamos a través de imágenes digitales obtenidas gracias a escáneres o cámaras digitales. Por este motivo explicaremos a continuación los conceptos básicos relacionados con las imágenes digitales, pudiendo encontrar desarrollos en más detalle en [1, Cap. 3].

Definición

Podemos interpretar el concepto de imagen como una función bidimensional de la forma:

$$f(x, y) = i(x, y)r(x, y) \tag{2.1}$$

donde $i(x, y)$ es la cantidad de luz incidente sobre la escena (iluminación) y $r(x, y)$ es la parte de dicha luz que se refleja (reflexión). $i(x, y)$ depende de la fuente de luz existente, mientras que $r(x, y)$ depende del tipo de objeto en la escena.

Digitalización

En nuestro caso, trabajaremos con imágenes digitales: la digitalización de las coordenadas espaciales representa el concepto de muestreo, ya que de cada una de las dos

dimensiones de la imagen (continuas) tomamos un número determinado (discreto) de valores (píxeles); el nivel de gris (en imágenes en blanco y negro) representa la digitalización de la amplitud de la señal (continua). Por lo que podemos decir que la función $f(x, y)$ es discreta tanto en dominio como en rango. De esta forma, podemos entender una imagen digital como una matriz de dimensiones N x M, en la que sus elementos representan cada uno de los píxeles de la imagen y cuyo valor $f(i, j)$ refleja su nivel de gris, de la forma:

$$f(x,y) = \begin{pmatrix} f(0,0) & f(1,0) & \cdots & f(N-2,0) & f(N-1,0) \\ f(0,1) & f(1,1) & \cdots & f(N-2,1) & f(N-1,1) \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ f(0,M-2) & f(1,M-2) & \cdots & f(N-2,M-2) & f(N-1,M-2) \\ f(0,M-1) & f(1,M-1) & \cdots & f(N-2,M-1) & f(N-1,M-1) \end{pmatrix}$$

Como hemos introducido antes, el muestreo de las dos dimensiones del espacio nos da cada uno de los píxeles de la imagen digital. Así, con un número muy elevado de píxeles tendremos una imagen digital más cercana a la realidad. Cuanto menor sea el número de píxeles, más información de la imagen real estaremos perdiendo y por tanto peor será la imagen digital. De la misma forma, el valor que toma la función $f(x, y)$ en cada punto es teóricamente cualquiera entre 0 e infinito. Debemos digitalizar ese rango de valores para poder representarlo, por lo que cuantificamos en un número determinado de niveles de gris. En el caso de usar 8 bits, existen 256 niveles distintos, siendo 0 el valor correspondiente a un píxel sobre el que no incide luz o absorbe toda la que recibe y 255 el correspondiente a un píxel sobre el que incide una luz infinita.

Resolución

Relacionado con el muestreo y la cuantificación aparece el concepto de resolución de la imagen. Podemos definirlo como el nivel de detalle que se aprecia en la imagen. A mayor resolución deseada, más píxeles y niveles de gris necesitaremos en la imagen. Dependiendo de la aplicación en la que nos encontremos será necesario un muestreo espacial u otro. Por ejemplo, si intentamos buscar objetos de pequeño tamaño en una imagen, el muestreo debe ser elevado para no perder los detalles. En cambio, si únicamente buscamos zonas de la imagen, podremos usar un menor número de muestras.

En cuanto al número de niveles de gris, el ojo humano distingue en torno a 25, por lo que aparentemente no encontraremos diferencias, pero con un número pequeño de

Figura 2.1: Imagen ejemplo 300 dpi

Figura 2.2: Imagen ejemplo 72 dpi

niveles de gris aparece el fenómeno de falsos contornos.

Histograma

Una de las herramientas más empleadas a la hora de trabajar con imágenes es el histograma. Se trata de una función que devuelve el número de veces que se repite un nivel de gris en la imagen. De esta forma obtenemos información útil con un conjunto menor de valores, pero tiene un gran inconveniente: no conocemos la localización de los píxeles, por lo que pueden existir imágenes completamente distintas con el mismo histograma.

Figura 2.3: Histogramas

Planos de bits

La representación de planos de bits de una imagen consigue reflejar la influencia del ruido. Cada píxel en la imagen tiene un valor de gris representado por n bits. El plano de bits i da como resultado una imagen binaria que toma para cada píxel únicamente el valor de su i-ésimo bit. Para planos de bits bajos la imagen tiene un aspecto de distribución aleatoria de puntos.

Vecindad y conectividad

La vecindad de un píxel p es el conjunto de píxeles que ocupan las posiciones adyacentes al mismo. Si las coordenadas de p son (x, y), los píxeles $(x-1, y)$, $(x+1, y)$, $(x, y-1)$ y $(x, y+1)$ forman la vecindad tipo 4 de p ($N4(p)$). Junto a los vecinos asociados a las diagonales $(x-1, y+1)$, $(x-1, y-1)$, $(x+1, y+1)$ y $(x+1, y-1)$ ($ND(p)$) forman los ocho vecinos de p ($N8(p)$). La conectividad expresa que dos píxeles pertenezcan al mismo objeto. Dos píxeles están conectados si son vecinos y sus niveles de gris cumplen alguna restricción (ser similares, por ejemplo). Dados dos píxeles p y q podemos hablar de: conectividad-4, si q pertenece a $N4(p)$; conectividad-8, si q pertenece a $N8(p)$; conectividad-m, si q pertenece a $N4(p)$ o q pertenece a $ND(p)$ y la intersección de los vecinos tipo 4 de p y q es el conjunto vacío. La conectividad-m (o mixta) elimina posibles conexiones múltiples entre píxeles, que harían que detectáramos más objetos de los que realmente hay en una imagen.

Distancia

Distancia es el mínimo número de pasos elementales que se necesitan para ir de un punto a otro. Una función de distancia D cumple:

- $D(p, q) \geq 0$, ($D(p, q) = 0$, si $p = q$)
- $D(p, q) = D(q, p)$
- $D(p, z) \leq D(p, q) + D(q, z)$

siendo p, q, y z píxeles de coordenadas (x, y), (s, t) y (u, v).

Existen distintas funciones de distancia, pero las más comunes son:

- *Distancia euclídea* La distancia euclídea entre p y q se define como $D_E(p, q) = \sqrt{(x-s)^2 + (y-t)^2}$.
- *Distancia Manhattan* $D = |x-s| + |y-t|$

- *Distancia tablero de ajedrez* $D(p,q) = max(x-s, y-t)$

La distancia euclídea es exacta, pero el coste computacional es elevado y no tiene en cuenta el concepto de vecindad. Según la definición de distancia Manhattan, los vecinos tipo 4 están a distancia unidad; en la distancia tablero de ajedrez son los vecinos tipo 8 los que se encuentran a distancia unidad.

Color

Para el ser humano el color es uno de los atributos más importantes de un objeto, pero en el ámbito de las imágenes digitales suponía un coste computacional excesivo hasta hace poco tiempo. Actualmente no hay problemas a la hora de trabajar con imágenes en color debido al hardware existente. Algunos conceptos relativos al color son:

Brillo: sensación que indica si un área está mas o menos iluminada.
Tono: sensación que indica si un área se asimila al rojo, amarillo, verde o azul o a una combinación de dos de ellos.
Coloración: sensación por la que un área tiene un mayor o menor tono.
Luminosidad: brillo de una zona respecto a otra blanca en la imagen.
Croma: la coloración de un área respecto al brillo de un blanco de referencia.
Saturación: la relación entre la coloración y el brillo.

Las distintas maneras de transformar los parámetros cromáticos en eléctricos y representar los colores son lo que conocemos como espacios de color. En general, un espacio de color es un modelo con el que se intenta describir la percepción humana que se conoce como color. En un espacio de color propiamente dicho se deben poder establecer relaciones entre los distintos colores (independientemente de sus intensidades, saturaciones, etc...).

Espacio RGB Se basa en la combinación del rojo, el verde y el azul, 3 señales de luminancia cromática distinta. Los colores se obtienen mediante la suma de las 3 componentes: $X = R + G + B$.

En la figura 2.4 podemos ver la representación del espacio de color RGB. En la diagonal del cubo se encuentran los niveles de gris, donde las tres componentes son iguales. Hasta ahora cada píxel tenía un valor de nivel de gris; en este espacio, cada píxel tiene tres valores, por lo que es necesario el triple de memoria para almacenarlo. Este

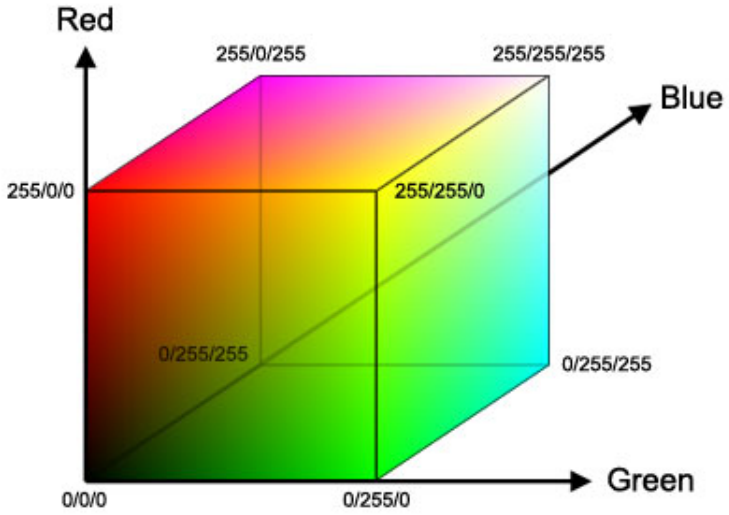

Figura 2.4: Espacio de color RGB

espacio tiene el problema de mezclar información relativa al color (tono y saturación) y a la intensidad en cada uno de sus tres valores. Este inconveniente se resuelve con el espacio de color HSI.

Espacio HSI Este espacio de color se basa en la manera que tenemos los humanos de percibir los colores. Utiliza tres componentes: tono (hue), saturación (saturation) y brillo (intensity); estos aspectos favorecen posibles segmentaciones en función del color. Podemos pasar del espacio RGB al HSI mediante las ecuaciones:

$$I = \frac{R+G+B}{3} \tag{2.2}$$

$$H = \arctan\left(\frac{\sqrt{3}(G-B)}{(R-G)+(R-B)}\right) \tag{2.3}$$

$$S = 1 - \frac{min(R,G,B)}{I} \tag{2.4}$$

$$H = \arctan\frac{x}{y} \tag{2.5}$$

En la figura siguiente vemos la representación del espacio de color HSI. La altura del corte con las pirámides (o cono) nos indica el valor de la intensidad I; la componente H viene dada por la orientación del color (tomando la referencia de 0^o en el rojo);

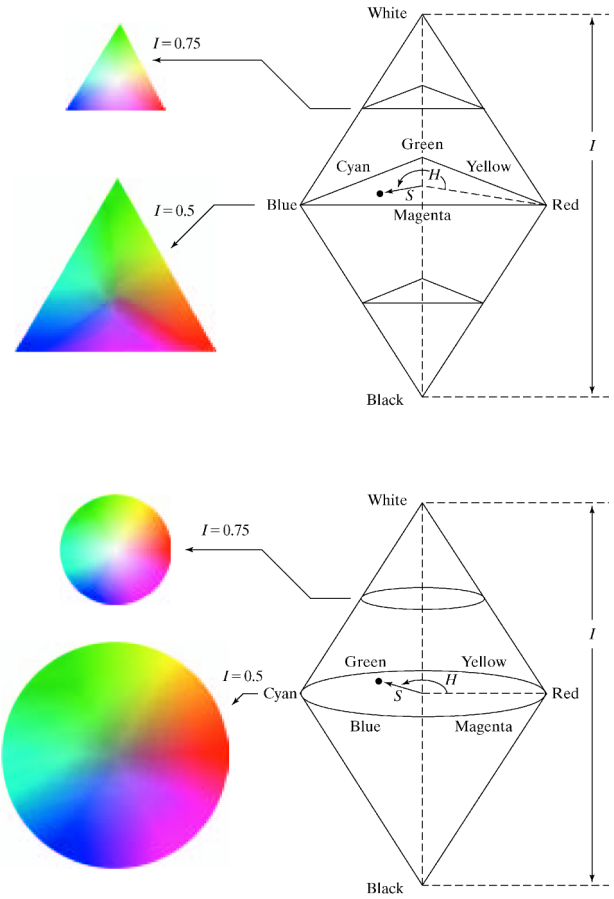

Figura 2.5: Espacio de color HSI

y la saturación S está representada por la distancia al centro del triángulo (o círculo).

Existen variaciones de este espacio de color como HSL (Hue Saturation Lightness), HSV (Hue Saturation Value), HCI (Hue Chroma/Colourfulness Lightness), HVC (Hue Value Chroma) o TSD (Hue Saturation Darkness).

Podemos encontrar dos inconvenientes en el uso del espacio HSI: el primero es su alto coste computacional debido a la no linealidad de las transformaciones necesarias; el segundo es la inestabilidad de los valores para el tono cuando $R = G = B$ ($S = 0$). La principal ventaja es la separación de aspectos relevantes como el tono y la luminosidad. De esta forma podremos tomar la componente H y distinguir entre dos objetos de distinto tono según su histograma; e igualmente podremos modificar la luminosidad de la imagen alterando la componente I, sin perjudicar el tono.

Además, existen otros espacios (*XYZ*, *Luv*, *Lab*, *YIQ*, *YUV*, *YCbCr*, *YCC*, *CMY(K)*...) usados en distintos ámbitos según sus características.

2.2. Visión artificial

2.2.1. Introducción a la visión artificial

En primer lugar, debemos definir el concepto sobre el que versa este proyecto: podemos llamar visión artificial al conjunto de técnicas que permiten obtener información a partir de imágenes tomadas por una cámara. Esa información puede ser de muchos tipos y con distintos objetivos. Por ejemplo, en control de calidad, la información que buscamos obtener es cuál de las piezas que estamos verificando es defectuosa, para lo que nos servimos de distintas transformaciones sobre las imágenes para llegar a esa conclusión.

Podemos establecer una cronología de la visión artificial:

- 1960 - Inicios del procesamiento digital de imágenes por computador.
- 1968 - Primera publicación sobre el tema: "Pattern Recognition"(Pergamon, ahora Elsevier).
- 1969 - Primer libro de texto: "Picture Processing by Computer"(A. Rosenfeld).
- 1970 - Primera conferencia internacional sobre reconocimiento de patrones (ICPR).
- 1977 - Creación del CVPR (Computer Vision and Pattern Recognition).
- 1978 - International Association for Pattern Recognition (IAPR).
- 1979 - Artículos del IEEE sobre Análisis de patrones e inteligencia de máquinas (PAMI).
- 1987 - Primera Conferencia Internacional sobre Visión por Computador (ICCV).

Podemos encontrar visión artificial en control de procesos, detección de eventos (por ejemplo detección de personas en cámaras de vigilancia), robótica, etc. En nuestro caso, nos ocupa el reconocimiento óptico de caracteres (optical character recognition, OCR). Para realizar esta tarea, existen varias librerías de funciones que nos permiten programar aplicaciones relativamente sencillas de visión por computador, pero también potentes. Algunas de ellas son las librerías MIL (Matrox Imaging Library), OpenCV, PIL (Python Imaging Library) y otras muchas. Nos centraremos en las utilizadas en este proyecto, las librerías de OpenCV.

OpenCV es una librería de código abierto para trabajar con visión artificial. Código abierto significa que su distribución y desarrollo es completamente libre (no confundirlo con software libre, donde el usuario tiene los mismos derechos, pero una vez adquirido el producto). La principal ventaja del software de código abierto es su evolución, gracias a las contribuciones de todos sus usuarios. Estas librerías están escritas en C y C++ y se pueden utilizar bajo Linux (nuestro caso), Windows y Mac OS X, aunque se están desarrollando interfaces para Python, Ruby o Matlab. OpenCV nació de una investigación desarrollada por Intel al decidir que era necesario disponer de un conjunto de funciones básicas como inicio de todo nuevo proyecto emprendido por estudiantes o programadores. OpenCV busca ser computacionalmente eficaz, ya que es uno de los factores básicos en las aplicaciones en tiempo real. Los tres objetivos principales de OpenCV son:

- Avanzar en la investigación de la visión artificial proporcionando un software abierto y al mismo tiempo optimizado
- Difundir el conocimiento de la visión artificial mediante una infraestructura común, con la cual el código sea entendible por cualquier desarrollador
- Conseguir mejores aplicaciones comerciales basadas en la visión artificial por el hecho de existir un código optimizado accesible a todos

En los anexos explicaremos detalladamente las funciones de OpenCV utilizadas en el proyecto.

2.2.2. Preprocesamiento de imágenes

Como ya hemos mencionado, el preprocesamiento de imágenes tiene por objetivo conseguir la imagen más adecuada posible a nuestros intereses, no significando esto conseguir la imagen más nítida, por ejemplo. En nuestro caso, el propósito es extraer cada uno de los caracteres para su posterior reconocimiento. Esto no implica obtener una imagen "que se vea mejor", sino una imagen con una información más aprovechable. El uso de estos algoritmos de preprocesamiento depende únicamente del objetivo que busquemos, por lo que en algunas aplicaciones pueden ser imprescindibles y en otras incluso perjudiciales. A continuación veremos los principales algoritmos de preprocesamiento de imágenes, sus utilidades y su posible aplicación a nuestro caso. Las expresiones utilizadas en los siguientes apartados se encuentran en [1, Cap. 5].

Manipulación del contraste

El contraste de una imagen se puede definir como la diferencia local de intensidad entre sus píxeles. En nuestro caso trabajaremos con imágenes en blanco y negro, por lo que la intensidad viene dada por el nivel de gris de cada píxel. En imágenes supuestas ideales, la iluminación es uniforme y la relación entre la luz de entrada y la imagen final es lineal. Estas dos afirmaciones implicarían que la diferencia entre dos píxeles de la imagen se debe únicamente a diferencias en su valor y no a cambios de luminosidad ni a ganancias no lineales. En la realidad no ocurre así, y podemos encontrar píxeles en la imagen final saturados (con un nivel de gris máximo sin serlo en realidad) o con un valor nulo sin ser su entrada cero. Mediante el histograma de la imagen podremos distinguir entre dos casos: que los niveles de gris estén concentrados en un pequeño intervalo, o que tomen todos los posibles valores; dependiendo del caso utilizaremos unos algoritmos u otros. Como ya veremos más adelante, con los textos antiguos con los que trabajamos en este proyecto no será necesario utilizar estos algoritmos, ya que las imágenes están tomadas en situaciones controladas de iluminación.

Amplitud de la escala Dado el histograma de una imagen, la amplitud de la escala es útil cuándo éste toma valores comprendidos en un pequeño intervalo de niveles de gris. Creamos una función que redistribuye los píxeles en todo el rango posible de niveles de gris (en nuestro caso 256 ya que trabajamos con imágenes de 8 bits) de la forma:

$$y = T(c) = A\frac{c-a}{b-a} \tag{2.6}$$

donde:
a y b son los límites inferior y superior de niveles de gris en el histograma.
c es el nivel de gris de la imagen inicial.
A es el valor máximo que puede tomar un píxel (255).

Con esta transformación logramos una mayor separación entre los niveles de gris presentes en la imagen inicial, por lo que podemos distinguir mejor los detalles. Con la amplitud de la escala la información presente en la imagen es la misma, pero se favorece una posible detección de bordes posterior y se logra cierta independencia de la iluminación.

Modificación del contraste Al igual que en el caso anterior, modificamos el nivel de gris de cada píxel, pero utilizamos otro tipo de funciones de la forma $p = m^a$ (siendo p y m los niveles de gris de la imagen final e inicial respectivamente, y a la potencia a

la que se eleva), como las siguientes:
Función inversa

$$p = 255 - m \tag{2.7}$$

Función cuadrada

$$p = \frac{m^2}{255} \tag{2.8}$$

Función cúbica

$$p = \frac{m^3}{255^3} \tag{2.9}$$

Función raíz cuadrada

$$p = \sqrt{255m} \tag{2.10}$$

Función raíz cúbica

$$p = \sqrt[3]{255^2 m} \tag{2.11}$$

Función logarítmica

$$p = 255\frac{ln(1+m)}{ln(1+255)} \tag{2.12}$$

Estas expresiones están particularizadas para el caso de una imagen de 8 bits, ya que aparece como valor máximo de nivel de gris 255. A diferencia de la amplitud de escala, estas transformaciones son no lineales y se beneficia a las zonas oscuras en detrimento de las claras, o viceversa. Si se quiere modificar la relación entre los niveles centrales respecto a los extremos, utilizamos funciones de la forma:

$$f(x) = \frac{255}{2}\left(1 + \frac{1}{\sin\left(\alpha\frac{\pi}{2}\right)}\sin\left(\alpha\pi\left(\frac{x}{255} - \frac{1}{2}\right)\right)\right) \tag{2.13}$$

ó

$$f(x) = \frac{255}{2}\left(1 + \frac{1}{\tan\left(\alpha\frac{\pi}{2}\right)}\tan\left(\alpha\pi\left(\frac{x}{255} - \frac{1}{2}\right)\right)\right) \tag{2.14}$$

La ecuación 2.13 favorece a los valores medios respecto a los claros y oscuros, y la 2.14 consigue el efecto contrario. Ambas son funciones sigmoides, ya que tienen forma de S. Este tipo de funciones también se emplean como funciones de activación en las redes neuronales, de lo que hablaremos en el apartado correspondiente.

Modificación del histograma Los métodos anteriores trabajan sobre cada uno de los píxeles de la imagen modificando su nivel de gris. En la modificación del histograma se busca alterar la forma del histograma completo, por lo que se dice que es un método global. La manera más habitual de modificarlo es mediante su ecualización, es decir, conseguir que el número de píxeles de cada nivel de gris en la imagen sea el mismo, o lo más parecido posible. Para ello, trabajamos con el histograma acumulado de la imagen:

$$H(i) = \sum_{k=0}^{i} h(k) \tag{2.15}$$

siendo:
i un nivel de gris determinado.
$h(k)$ el número de píxeles en la imagen con un nivel de gris k.
Con un histograma plano, el histograma acumulado sería:

$$G(i') = (i' + 1)\frac{NM}{256} \tag{2.16}$$

donde:
N y M son las dimensiones de la imagen.
256 son los niveles de gris de la imagen (trabajamos con 8 bits).
El objetivo de la ecualización es que $H(i) = G(i')$, por lo que:

$$H(i) = (i' + 1)\frac{NM}{256} \tag{2.17}$$

Obteniendo:

$$i' = \frac{256}{NM}H(i) - 1 \tag{2.18}$$

De esta forma, a cada nivel de gris en la imagen original i le corresponde un nivel de gris i' en la imagen modificada, de forma que el histograma acumulado de ésta sea lo más plano posible. Dado que los niveles de gris deben ser valores enteros:

$$i' = \left[\frac{256}{NM}H(i) - 1\right] \tag{2.19}$$

siendo $int(x) = [x]$.

Esta modificación del histograma es útil únicamente con imágenes oscuras o claras en conjunto, pero no cuando la imagen tiene distintas tendencias. Para salvar este inconveniente, se ha desarrollado una ecualización por ventanas, que consiste en realizar la ecualización de una porción de la imagen. Se centra la ventana en un píxel, se ecualiza,

y se sustituye el valor del píxel central por el resultante después de la ecualización según la ecuación 2.19. Existen otras formas de ecualizar el histograma aparte de intentar conseguir un histograma "plano", como usar distribuciones de tipo:

- Exponencial
- Rayleigh
- Raíz cúbica
- Logaritmo

Tablas de consulta Todas estas operaciones requieren un coste computacional alto, ya que constan de operaciones como multiplicaciones, logaritmos, etc. que se deben aplicar a cada uno de los píxeles de la imagen. Una manera alternativa de obtener el mismo resultado es el uso de tablas de consulta o look up tables (LUTs). Se pueden imaginar como un vector de 256x1 (en imágenes de 8 bits) en el que el elemento i-ésimo corresponde al nivel de gris del píxel i en la imagen modificada. De esta forma los cálculos se realizan una sola vez al iniciar el algoritmo y para cada píxel únicamente hay que hacer una asignación.

Eliminación de ruido

Todas las imágenes, debido al sensor de la cámara o al medio de transmisión de la señal, presentan alteraciones en el nivel de gris de sus píxeles. Esas alteraciones se conocen como ruido de la imagen, está presente en las altas frecuencias y podemos clasificarlo como:

Gaussiano Este tipo de ruido hace que todos los píxeles varíen ligeramente su nivel de gris; se conoce como gaussiano porque esas variaciones siguen aproximadamente una distribución normal. Puede deberse a ruido en los digitalizadores, interferencias en la transmisión o diferentes ganancias en el sensor, por ejemplo.

Impulsional En lugar de causar una pequeña variación a todos los píxeles, provoca que ciertos píxeles de la imagen tomen valores muy altos (saturación) o se pierda su señal. Es conocido también como ruido sal y pimienta por este motivo.

Frecuencial La imagen que obtenemos es la suma de la imagen deseada y una interferencia que podemos caracterizar como una senoide de una frecuencia dada.

Multiplicativo Como su nombre indica, la imagen obtenida es la multiplicación de dos señales.

Para disminuir el efecto del ruido en nuestras imágenes, existen principalmente tres tipos de filtros:

- Filtros lineales espaciales
- Filtros no lineales
- Filtros en el dominio de la frecuencia

A continuación se explicarán las características básicas de cada uno de ellos.

Filtros lineales espaciales

Suma de imágenes El ruido afecta a los píxeles de manera aleatoria en el espacio, es decir, un píxel en particular puede verse modificado en mayor o menor medida si tomamos la imagen en distintos instantes de tiempo. Por este motivo, una forma de corregir el efecto del ruido gaussiano o impulsional es tomar un número n de imágenes y realizar la media (filtro paso bajo temporal). Al utilizar distintos instantes de tiempo, no es recomendable este algoritmo para imágenes en movimiento.

Filtros paso bajo espaciales El ruido se encuentra en las frecuencias altas, por lo que necesitamos filtros paso bajo para eliminarlo. Utilizando un filtro de 3x3 formado por unos en todas sus posiciones y con un coeficiente de $\frac{1}{9}$, obtenemos la media de sus píxeles, eliminando en cierta medida el ruido empleando una sola imagen. A partir de este filtro compuesto por unos, en el que ponderamos todos los píxeles de la máscara por igual, podemos definir otros que den más importancia al central que a los vecinos tipo 4 o tipo 8.

1	1	1
1	1	1
1	1	1

1	1	1
1	2	1
1	1	1

1	2	1
2	4	2
1	2	1

En general,

1	b	1
b	b^2	b
1	b	1

En todos los casos la ganancia del filtro debe ser la unidad, ya que en caso contrario estaríamos desvirtuando la imagen. Estos filtros obtienen un buen resultado eliminando ruido gaussiano, pero al tratarse de filtros paso bajo, perjudican también los contornos y los objetos pequeños en la imagen (representados por altas frecuencias). Además, estos filtros tampoco eliminan eficazmente el ruido impulsional, ya que los píxeles afectados siguen presentando niveles de gris menores, pero aún así elevados; sus vecinos también se ven influidos por el valor extremo del ruido impulsional, ya que al hacer la media con él aumentan su valor.

Gaussiana Estas máscaras imitan la distribución gaussiana:

$$G(x, y) = \exp\left(-\frac{(x+y)^2}{2\sigma^2}\right) \tag{2.20}$$

Presentan los mismos problemas que los filtros paso bajo espaciales.

Filtros no lineales

Outlier El algoritmo es el siguiente: tomamos para cada píxel sus ocho vecinos y comparamos su valor con la media de éstos. Si la diferencia es mayor que un umbral dado, sustituimos su valor por la media de sus vecinos. De esta forma mejoramos ligeramente la respuesta del filtro ante ruido impulsional, aunque, igual que ocurría con los filtros paso bajo espaciales, la influencia de los valores extremos en la media perjudica los resultados.

Mediana Hemos visto que el ruido impulsional afecta de manera negativa en la media, por lo que no debemos usarla. La solución es emplear la mediana, un operador que consiste en tomar un elemento de una secuencia ordenada de N valores, tal que existan $\frac{(N-1)}{2}$ valores menores o iguales que él y $\frac{(N-1)}{2}$ valores mayores o iguales que él. De esta forma, en todas las secuencias de valores que tomemos, los píxeles correspondientes a ruido impulsional ocuparán las posiciones extremas, por lo que serán descartados. Obtenemos un buen resultado eliminando el ruido impulsional y una mejor preservación de bordes. El principal inconveniente de este algoritmo es su velocidad,

ya que la ordenación de los valores de los vecinos es bastante lenta en comparación a la media.

Filtros en el dominio de la frecuencia

Filtros paso bajo Atenúan (idealmente eliminan) las frecuencias por encima de la frecuencia de corte. Un filtro paso bajo ideal tampoco sería recomendable, ya que en las altas frecuencias se encuentra el ruido, pero también los bordes, que no debemos eliminar. Se busca un compromiso entre una atenuación suficiente del ruido, sin perjudicar en exceso los bordes.

Filtros de Butterworth Su principal ventaja es que la ganancia en la zona de interés es muy uniforme, no favoreciendo a unas frecuencias más que a otras.

Otros tipos de filtros Se pueden crear filtros acordes a nuestros intereses: paso bajo o paso alto a una frecuencia de corte deseada ω_0, paso banda, rechazo de banda... Son más lentos computacionalmente respecto a los espaciales, pero tienen la ventaja de ser mucho más flexibles, ya que podemos corregir interferencias a casi cualquier frecuencia.

Filtro homomórfico Es el filtro empleado cuando queremos eliminar el ruido multiplicativo. Se considera la imagen como producto de una señal libre de ruido por otra señal que interfiere. Si tomamos logaritmos, convertimos el producto en una suma; y si tomamos transformadas de Fourier conseguimos una situación análoga a las estudiadas anteriormente, que podemos corregir con un filtro paso bajo y una posterior antitransformada y deshaciendo el logaritmo. Podemos ver estos pasos en las ecuaciones siguientes:

$$f(x,y) = r(x,y)i(x,y) \tag{2.21}$$

$$z(x,y) = \ln f(x,y) = \ln r(x,y) + \ln i(x,y) \tag{2.22}$$

$$F(z(x,y)) = Z(u,v) = F(\ln r(x,y) + \ln i(x,y)) = R(u,v) + I(u,v) \tag{2.23}$$

Realce de bordes

Como ya hemos mencionado, los contornos de la imagen se encuentran en las altas frecuencias, al igual que el ruido. Por tanto si queremos realzar bordes aumentaremos el efecto del ruido, por lo que primero eliminaremos el ruido y posteriormente realzaremos los contornos.

Filtros lineales en el dominio del espacio El efecto que se busca es el siquiente:

$$\text{Imagen resultante} = \text{Ganancia} \cdot \text{Imagen original - Bajas frecuencias}$$

De esta forma aumentamos las altas frecuencias de la imagen original y quitamos las bajas. La máscara empleada para atenuar las altas frecuencias estaba compuesta por unos, por lo que para conseguir el efecto contrario tendremos:

$$\frac{1}{9}\begin{array}{|c|c|c|}\hline -1 & -1 & -1 \\ \hline -1 & A & -1 \\ \hline -1 & -1 & -1 \\ \hline \end{array}$$

donde:
$A = 9$ Ganancia - 1

Otra manera de conseguir este efecto es, en lugar de quitar las bajas frecuencias de la imagen original, sumar las altas frecuencias.

$$\text{Imagen resultante} = \text{Imagen original} + \text{Altas frecuencias}$$

Usando la laplaciana, que expresa la derivada en todas direcciones, podemos obtener otros tipos de filtro paso alto.

-1	-1	-1
-1	8	-1
-1	-1	-1

Al igual que ocurría en los filtros paso bajo, tendremos distintas máscaras en función de la importancia que le demos al píxel central respecto a sus vecinos, como por ejemplo:

0	-1	0
-1	4	-1
0	-1	0

-1	-2	-1
-2	12	-2
-1	-2	-1

También podemos modificar el valor del píxel central para conseguir una ganancia unidad.

Filtros lineales en el dominio de la frecuencia Como vimos en el apartado de reducción de ruido, se obtiene la transformada de Fourier de la imagen original, pero aplicándose ahora un filtro paso alto similar a los analógicos, por lo que aislamos las altas frecuencias; finalmente se suman a la imagen original. Este método es mucho más flexible que las convoluciones, pero con mayor coste computacional.

Filtros no lineales: filtro max-min Desarrollado por Kramer y Bruchner, obtiene los valores máximos y mínimos en el entorno de un píxel, siendo el nuevo valor:

$$g(x,y) = \begin{cases} f_{max} & si \quad f_{max} - f(x,y) \leq f(x,y) - f_{min} \\ f_{min} & en \quad caso \quad contrario \end{cases} \tag{2.24}$$

Falso color Este algoritmo tiene como objetivo facilitar un posterior análisis humano de las imágenes. El ojo humano sólo distingue una gama de 16 niveles de gris, mientras que puede diferenciar un número mucho mayor de colores. Suele haber dos casos típicos: en el primero se busca mejorar el contraste entre los grises y en el segundo resaltar algunos niveles de gris respecto a los demás. En ambos casos usamos LUTs para su implementación.

2.2.3. Extracción de características

El siguiente paso al procesado de la imagen es la extracción de características, que consiste en detectar los objetos presentes en ella. Existen algoritmos para la detección de bordes, líneas, esquinas y texturas. Se intentan encontrar en la imagen las características que definen el objeto buscado. Podemos encontrar un análisis más detallado y el origen de las expresiones empleadas aquí en [1, Cap. 6].

Detección de bordes

Los bordes definen la frontera entre lo que consideramos objeto y fondo, o entre distintos objetos. Por este motivo constituyen una información muy útil. Además, pueden ser la etapa inicial de algoritmos de segmentación. Los detectores de bordes se basan en encontrar los puntos de la imagen en los que existe una variación brusca en el nivel de gris. Estamos buscando por tanto incrementos, por lo que el operador que utilizaremos será el operador derivada. Podremos usar el gradiente (primera derivada)

o la laplaciana (segunda derivada), según busquemos máximos y mínimos o pasos por cero. Como hemos visto en apartados anteriores, en las imágenes digitales existe ruido. Este ruido puede hacer que consideremos como borde aquello que no lo es, por lo que es necesario umbralizar el nivel de gris de un píxel para definirlo como borde o no. En el caso de emplear el gradiente, impondremos la condición de que los bordes posean más de un píxel de grosor. De esta forma, aunque conseguimos bordes con menor precisión, eliminamos la influencia del ruido. Utilizando la laplaciana no es necesario este procedimiento, aunque computacionalmente es un método más costoso y lento. Dependiendo de las restricciones que tengamos, elegiremos uno u otro.

Técnicas basadas en el gradiente La expresión general del operador gradiente aplicado sobre una imagen $f(x, y)$ es:

$$\nabla f(x,y) = [G_x \quad G_y] = \left[\frac{\partial f}{\partial x} \quad \frac{\partial f}{\partial y}\right] \tag{2.25}$$

Este vector gradiente representa la dirección de mayor variación de intensidad para el punto (x, y). Podemos definir ese vector por su módulo y dirección:

$$|\nabla f| = \sqrt{G_x^2 + G_y^2} \tag{2.26}$$

$$\angle \nabla f = \arctan\left(\frac{G_y}{G_x}\right) \tag{2.27}$$

La expresión del módulo a menudo se cambia por la siguiente, que exige un menor coste computacional:

$$|\nabla f| = |G_x| + |G_y| \tag{2.28}$$

Al tratarse de derivadas en imágenes digitales, con valores discretos, la expresión general del gradiente incluirá incrementos, de la forma:

$$\nabla f(x,y) = [G_x \quad G_y] = \left[\frac{\Delta f}{\Delta x} \quad \frac{\Delta f}{\Delta y}\right] \tag{2.29}$$

De esta forma podemos interpretar el gradiente como el cálculo de diferencias entre píxeles vecinos: según qué píxeles consideremos, tendremos derivadas unidimensionales o bidimensionales, aplicadas a una dirección determinada o en todas las direcciones de forma global, etc.

El gradiente en la dirección x se puede representar por la máscara:

$$\begin{array}{|c|c|}\hline -1 & 1 \\ \hline \end{array} * f(x,y)$$

y en la dirección y:

$$\begin{array}{|c|}\hline -1 \\ \hline 1 \\ \hline \end{array} * f(x,y)$$

Estas máscaras tienen el problema de verse afectadas por el ruido, ya que únicamente tienen en cuenta dos píxeles. Existen otros filtros que además de calcular el gradiente tienen efectos de suavizado (filtros paso bajo). A continuación veremos algunos de ellos.

Operador de Roberts

0	-1
1	0

-1	0
0	1

Operador de Prewitt

-1	0	1
-1	0	1
-1	0	1

-1	-1	-1
0	0	0
1	1	1

Operador de Sobel

-1	0	1
-2	0	2
-1	0	1

-1	-2	-1
0	0	0
1	2	1

Operador isotrópico

-1	0	1
$-\sqrt{2}$	0	$\sqrt{2}$
-1	0	1

-1	$-\sqrt{2}$	-1
0	0	0
1	$\sqrt{2}$	1

También llamado operador de Frei-Chen.

La filosofía de todos ellos es la misma, combinar el gradiente con elementos similares a los filtros para reducir el ruido, pero con diferencias como las dimensiones de la máscara (a mayor dimensión, menor influencia del ruido), importancia dada a unos píxeles u otros (Sobel mayor importancia a píxeles centrales que Prewitt), etc. El operador de Roberts es el menos utilizado, ya que produce peores resultados al usar un entorno de 2x2. Entre los tres operadores restantes existen ligeros matices que los diferencian (Prewitt detecta mejor bordes verticales, Sobel diagonales e isotrópico intenta buscar el equilibrio), pero todos ellos dan un resultado similar. Por costumbre el más utilizado es el operador de Sobel.

Operadores de segundo orden Estos operadores tienen en cuenta la laplaciana:

$$\nabla^2 f(x,y) = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} \tag{2.30}$$

Cuyas máscaras más frecuentes son:

-1	-1	-1
-1	8	-1
-1	-1	-1

0	-1	0
-1	4	-1
0	-1	0

Este operador es muy sensible al ruido, por lo que únicamente se usa como paso intermedio para el operador de Marr-Hildreth, que veremos a continuación.

Métodos basados en las derivadas de gaussianas

El operador de Marr-Hildreth Marr-Hildreth impone dos condiciones al filtro que se utilice:

- El filtro debe ser local y tener en cuenta únicamente la información de los puntos cercanos al de estudio, ya que se deben detectar los bordes.

- Debe reducir el número de frecuencias que utiliza, por lo que debe ser un filtro paso-banda.

Consiste en convolucionar la imagen con la laplaciana de una gaussiana. La expresión general de dicha gaussiana será:

$$G(x,y) = \exp\left(-\frac{(x+y)^2}{2\sigma^2}\right) \tag{2.31}$$

Si se convoluciona la imagen con esta gaussiana disminuirá la influencia del ruido. Si calculamos la laplaciana de la ecuación 2.31 (es decir, su segunda derivada), obtenemos el operador de Marr-Hildreth:

$$H(x,y) = \nabla^2(G(x,y) * I(x,y)) \tag{2.32}$$

Por propiedades de la gaussiana, podemos pasar a esta ecuación:

$$H(x,y) = (\nabla^2 G(x,y)) * I(x,y) \tag{2.33}$$

siendo

$$\nabla^2 G(x,y) = \frac{r^2 - 2\sigma^2}{\sigma^4} \exp\left(\frac{-r^2}{2\sigma^2}\right); r^2 = x^2 + y^2 \tag{2.34}$$

Una vez se ha convolucionado la imagen con la laplaciana de la gaussiana, se encuentran los pasos por cero y obtendremos los bordes. Cuanto menor sea la desviación típica de la gaussiana (σ), mayor número de bordes aparecerán. Debemos tener cuidado en este aspecto, ya que una mala elección de dicha desviación puede dar lugar a lo que se conoce como falsos contornos, encontrar más bordes de los que realmente existen. Otra forma de obtener los bordes de una imagen es aprovechar el hecho de que la laplaciana de una gaussiana (LoG) es muy similar a la diferencia de dos gaussianas (DoG). Convolucionamos la imagen con esas dos gaussianas, las restamos y buscamos los pasos por cero.

Detector de Canny El objetivo de John F. Canny en 1986 era encontrar el algoritmo de detección de bordes óptimo, para lo que impuso tres condiciones:

- *Error* El algoritmo debe encontrar el mayor número posible de bordes reales en la imagen.
- *Localización* Un píxel marcado como borde debe estar lo más próximo posible al borde real.
- *Respuesta* El borde de una imagen sólo debe ser marcado una vez, y siempre que sea posible, el ruido de la imagen no debe generar falsos bordes.

Estas tres condiciones se expresan matemáticamente como:

$$SNR = \frac{A\left|\int_{-W}^{0} f(x)\,dx\right|}{n_0\sqrt{\int_{-W}^{W} f^2(x)\,dx}} \tag{2.35}$$

$$Localizacion = \frac{A|f(0)|}{n_0\sqrt{\int_{-W}^{W} f^2(x)\,dx}} \tag{2.36}$$

$$Distancia = \pi\left(\int_{-\infty}^{+\infty} f^2(x)\,dx + \int_{-\infty}^{+\infty} f'^2(x)\,dx\right)^{\frac{1}{2}} \tag{2.37}$$

En este caso se llega a la conclusión de que el operador óptimo es la derivada de una gaussiana. Para obtener los bordes se siguen estos pasos:

- Se tiene una imagen I.
- Se tiene una gaussiana G.

- Se obtienen las derivadas G_x y G_y.
- Se convoluciona la imagen con las derivadas, obteniendo I_x e I_y.
- Se obtiene la magnitud M como la raíz cuadrada de la suma de los cuadrados de I_x e I_y.
- Se suprimen aquellos píxeles que no sean máximos, comparando el valor de M para cada píxel con sus vecinos.
- Se detectan los bordes con dos umbrales $T1$ y $T2$. La condición para pertenecer a un borde es ser mayor que $T2$ o mayor que $T1$ siempre que uno de sus vecinos sea mayor que $T2$.

Operadores de ajuste al modelo Estos operadores permiten medir el gradiente en determinadas direcciones (este, noreste, norte, noroeste...). Existen diferentes tipos de máscara, todas ellas de dimensión 3x3: Prewitt, Kirsch o Robinson.

Detección de líneas Podemos entender que la detección de líneas está incluida en lo ya visto para detección de bordes. Tenemos cuatro máscaras que detectan líneas horizontales, verticales, y diagonales a ± 45º.

E - O (0º)

-1	-1	-1
2	2	2
-1	-1	-1

NE - SO (45º)

-1	-1	2
-1	2	-1
2	-1	-1

N - S (90º)

-1	2	-1
-1	2	-1
2	-1	-1

NO - SE (-45º)

2	-1	-1
-1	2	-1
-1	-1	2

Detección de esquinas Existen dos motivos por los que encontramos esquinas en una imagen: el primero es la intersección de dos bordes de un objeto y el segundo es una discontinuidad en los niveles de gris debida a la textura de los objetos. Tomasi y Kanade proponen un método que consiste en determinar las zonas de la imagen con

elevados gradientes verticales y horizontales al mismo tiempo. Se analiza en el entorno de un punto la expresión:

$$C = \begin{bmatrix} \sum I_x^2 & \sum I_x I_y \\ \sum I_x I_y & \sum I_y^2 \end{bmatrix} \tag{2.38}$$

siendo I_x e I_y las derivadas parciales de I en dirección horizontal y vertical respectivamente. Esta matriz simétrica se puede diagonalizar a:

$$C = \begin{bmatrix} C_1 & 0 \\ 0 & C_2 \end{bmatrix} \tag{2.39}$$

Siendo C_1 y C_2 los autovalores de la matriz 2.38. Estos autovalores reflejan el valor del gradiente, cuya dirección viene dada por el autovector correspondiente. Para detectar las esquinas, habrá que calcular C_1 y C_2 y comprobar que el menor de ellos supera un umbral dado.

Otro método es basarnos en la variación de la dirección de los gradientes. Kitchen y Rosendfeld utilizan el producto de la curvatura horizontal por el gradiente de la imagen:

$$C = \frac{I_{xx} I_y^2 - 2 I_{xy} I_x I_y + I_{yy} I_x^2}{I_x^2 + I_y^2} \tag{2.40}$$

Ambos métodos no dan resultados idénticos, ya que dependen de factores como el entorno elegido en el primero o el umbral seleccionado en el segundo.

Texturas En algunos materiales, observamos que poseen un aspecto homogéneo aunque su nivel de gris no sea constante. Podemos definir una textura como un patrón visual homogéneo que se observa en un cierto tipo de material. Su estudio es importante porque puede ser una característica que defina un objeto determinado. Está íntimamente relacionado con la resolución de la imagen, ya que con un tamaño dado de imagen podemos encontrar texturas completamente distintas a las que existen con otro tamaño.

Para su estudio se siguen dos aproximaciones: estadística (estadísticos de primer y segundo orden: media, desviación típica, coeficiente de asimetría o tercer momento, apuntamiento o curtosis, entropía y matrices de coocurrencia) y frecuencial. En la primera se analizan valores estadísticos de los niveles de gris en un área, picos, valles u otra propiedad espacial. En la segunda, aparecerán picos en la transformada de Fourier a la frecuencia de repetición del patrón de la textura.

Detección de movimiento Los puntos del mismo objeto presentan velocidades similares, por lo que la detección de movimiento puede ayudar a la segmentación de la imagen. Existen cuatro tipos de movimientos:

- Movimiento de la cámara
- Movimiento de los objetos
- Cambios en la iluminación
- Cambios en la estructura, forma o tamaño del objeto

Tenemos también varias restricciones:

- No existen cambios de iluminación entre imágenes
- Suponemos la cámara fija, por tanto son los objetos los que se mueven
- Los objetos son rígidos, es decir, las velocidades de los puntos de un mismo objeto deben ser parecidas

Podemos distinguir tres métodos de detección de movimiento. El primero de ellos utiliza la diferencia de imágenes. Es un método sencillo e inmune a los cambios de iluminación (ya que el intervalo de tiempo entre imágenes suele ser pequeño), pero detecta cambios de dos tipos: la sección de fondo tapada por el objeto en la segunda imagen y la parte del fondo que estaba tapada en la primera imagen y aparece en la segunda; esto implica que no podemos saber hacia dónde se mueve el objeto. Por tanto podemos saber si existe algún movimiento en la imagen, pero no tenemos información sobre él.

Otro método que sí intenta obtener el campo de velocidades de la imagen busca una serie de características en la primera imagen (como bordes o esquinas) y encuentra su correspondencia en la segunda. La dificultad radica en saber con certeza si esas características encontradas en la segunda imagen son realmente las de la primera. Son métodos rápidos, pero dan poca información.

El tercer y último método busca obtener el flujo óptico (optical flow) de la imagen, es decir, un mapa denso del campo de velocidades. Se tienen en cuenta todos los píxeles de la imagen, por lo que será un método más lento. Se busca en el entorno de un punto cuál es la zona más parecida en la segunda imagen según sus variaciones espaciales y temporales. Presenta el problema de la apertura, esto es, la incapacidad de determinar exactamente la dirección y magnitud de un movimiento tomando únicamente una pequeña región de la imagen.

2.2.4. Transformaciones morfológicas

Consideramos transformaciones morfológicas aquellas que modifican la estructura o forma de los objetos de una imagen. Se pueden realizar en imágenes binarias (únicamente dos niveles de gris) o en imágenes con varios niveles de gris. Permiten acondicionar

la imagen según nuestros futuros objetivos y también eliminan el ruido que existe. Este tipo de operaciones se basa en la teoría de conjuntos, de la que veremos algunos conceptos:

- Inclusión: Y estará incluído en X si todo elemento de Y pertenece a X.

$$Y \subset X : \forall y \in Y \Rightarrow y \in X \tag{2.41}$$

- Complemento: el complemento de X, X^c, está formado por todos los elementos que no pertenecen a X.

- Unión: la unión de X e Y está formada por todos los elementos incluídos en uno de los dos.

$$X \cup Y = \{x| \quad x \in X \quad o \quad x \in Y\} \tag{2.42}$$

- Intersección: la intersección de dos conjuntos está formada por sus elementos comunes.

$$X \cap Y = (X^c \cup Y^c)^c \tag{2.43}$$

- Traslación: un conjunto X es trasladado por un vector v cuando cada uno de los elementos de X sufre esa traslación. El nuevo conjunto será Xv.

A cada transformación $\Psi(X)$ se le puede asociar una transformación dual $\Psi^*(X)$, tal que $\Psi^*(X) \rightarrow (\Psi(X^c))^c$.

Elemento estructural y transformaciones acierta o falla Los elementos estructurales son un conjunto de puntos que servirán para determinar la estructura de un conjunto X. Uno de ellos constituye el centro del elemento. Una transformación acierta o falla (del inglés hit or miss) se aplica sobre cada punto de un conjunto X de la siguiente forma:

- Del elemento estructural B se formará un conjunto B_x que a su vez formará otros dos conjuntos B_x^1 y B_x^2. B_x es el elemento estructural B desplazado para todo elemento $x \in X$. B_x^1 corresponde a los elementos del primer plano y B_x^2 a los del fondo.

- Un punto x pertenece a la transformación acierta o falla, $X \otimes B$, si y sólo si B_x^1 está incluido en X y B_x^2 está incluido en X^c:

$$X \otimes B = \{x| \quad B_x^1 \subset X; B_x^2 \subset X^c\} \tag{2.44}$$

Por ejemplo, si tenemos este elemento estructural:

0	1	0
1	**1**	1
0	1	0

El resultado de la transformación acierta o falla sólo será 1 en aquellos píxeles que tuvieran valor 1 en la imagen original y cuyos vecinos tipo 4 también lo tuvieran.

Erosiones y dilataciones Se tratan de las dos principales transformaciones morfológicas, cuya combinación da lugar a otras como la apertura y el cierre.

Erosión Esta transformación degrada progresivamente uno de los campos (0 o 1, en una imagen binaria), llegando a destruir la imagen en un número determinado de iteraciones. Si un píxel está rodeado de elementos iguales a él, seguirá perteneciendo al mismo campo; si no, pasará al otro campo. Matemáticamente:

$$X \ominus \breve{B} = \{x | B_x \subset X\} \tag{2.45}$$

siendo $\breve{B}$ el elemento estructural simétrico respecto al origen de B. Una erosión es análoga a una transformación acierta o falla donde B_x^2 es el conjunto vacío.

Dependiendo del tamaño del elemento estructural, la erosión será más notable o no. También podemos conseguir el efecto de utilizar un elemento estructural mayor realizando el proceso iterativamente, según la expresión $d = 1 + (n-1)N$, donde n es la dimensión del elemento estructural, N el número de iteraciones y d la dimensión equivalente.

Dilatación Al contrario que la erosión, la dilatación genera el crecimiento progresivo de uno de los campos. Un elemento del campo contrario a crecer pasará al campo que se expande si posee algún vecino que pertenezca a éste; en caso contrario, se mantiene igual. Análogamente al caso anterior, si se dilatara una imagen iterativamente llegaríamos a tener todos los píxeles de la imagen a nivel alto. La dilatación no es conmutativa con la erosión y viceversa. Gracias a este hecho aparecen la apertura y el cierre. Matemáticamente:

$$X \oplus \breve{B} = \left(X^c \ominus \breve{B}\right)^c \tag{2.46}$$

A continuación enunciaremos cuatro propiedades relativas a la dilatación y a la erosión:

$$(X \cup Y) \oplus \breve{B} = (X \oplus \breve{B}) \cup (Y \oplus \breve{B}) \tag{2.47}$$

$$(X \cap Y) \ominus \breve{B} = (X \ominus \breve{B}) \cap (Y \ominus \breve{B}) \tag{2.48}$$

$$(X \oplus \breve{B}1) \oplus \breve{B}2 = X \oplus (\breve{B}1 \oplus \breve{B}2) \tag{2.49}$$

$$(X \ominus \breve{B}1) \ominus \breve{B}2 = X \ominus (\breve{B}1 \ominus \breve{B}2) \tag{2.50}$$

Apertura y cierre

Apertura Como habíamos adelantado, las operaciones de erosión y dilatación no son conmutativas. Después de erosionar, no podemos recuperar la imagen original con una dilatación, pero obtenemos una nueva imagen con unas características interesantes. La apertura se define como una combinación de erosiones y dilataciones (primero erosión y finalmente dilatación) siempre con el mismo elemento estructural. Da como resultado una imagen con información de las partes más importantes de los objetos y con los elementos pequeños debidos al ruido eliminados.

$$X \circ B = (X \ominus \breve{B}) \oplus B \tag{2.51}$$

Cierre Operación dual de la apertura:

$$X \bullet B = (X \oplus \breve{B}) \ominus B \tag{2.52}$$

El cierre rellena pequeños agujeros en la imagen.

Otras transformaciones morfológicas Nos hemos centrado en las dos transformaciones más importantes, erosión y dilatación, y dos más que derivan de ellas, apertura y cierre, pero no son las únicas que existen. A continuación describiremos brevemente algunas otras transformaciones.

- *Esqueletización:* busca representar una región u objeto por su grafo, por ser útil para la representación y una de las características que definen un objeto. Además, pequeñas variaciones en la región modifican notablemente el esqueleto, por lo que es muy interesante en control de calidad. Existen varias formas de obtener el esqueleto de un objeto:
  - *transformación del eje medial*: para cada elemento de la región, buscamos el punto de borde más cercano.

- *máximos locales*: sustituimos cada píxel por la distancia menor al borde; es esqueleto vendrá dado por los máximos locales.

- *filtros morfológicos*: sucesión de transformaciones acierta o falla que van disminuyendo el tamaño de las regiones, con dos condiciones: no destruir los píxeles extremos y no romper la conectividad.

- *algoritmo de Zhang-Suen*: se eliminan aquellos píxeles que cumplan una serie de condiciones en función de sus 8 vecinos.

Estos algoritmos requieren un coste computacional elevado, por lo que es recomendable el uso de LUTs.

- *Adelgazamiento:* similar a la esqueletización, pero sin llegar al extremo. Suele usarse para "afinar" el resultado de una detección de bordes.

- *Extracción del perímetro:* el perímetro es una característica muy representativa de los objetos. La manera exacta de calcularlo requiere un coste computacional enorme (detectar un píxel del borde e ir moviéndonos por éste), por lo que buscamos otros métodos. Podemos calcular el perímetro interior como la diferencia de imágenes entre la original y la erosionada. Análogamente, el perímetro exterior vendrá dado por la resta de la imagen dilatada y la original. Únicamente tendríamos que acudir al histograma de la imagen para conocer el perímetro.

$$P_{interior}(A) = A - (A \ominus B) \tag{2.53}$$

$$P_{exterior}(A) = (A \oplus B) - A \tag{2.54}$$

- *Eliminación de ruido:* después de una segmentación se toman píxeles como pertenecientes al objeto cuando en realidad no lo son. Para los píxeles del fondo ocurre lo mismo. Para solucionarlo, realizamos combinaciones de erosiones y dilataciones.

- *Cerco convexo:* es la región más pequeña que contiene al objeto de tal forma que podemos unir dos puntos cualesquiera de dicho objeto con una línea recta y pertenecer todos los puntos de ella a la región. Se obtiene mediante un proceso opuesto a la esqueletización. Su diferencia con la imagen original da lugar a lagos y bahías, los cuales son muy útiles para clasificar los objetos de la imagen.

- *Segmentación mediante operador distancia:* si tenemos una imagen con objetos unidos, el etiquetado no será capaz de distinguirlos. Si aplicamos el operador distancia y umbralizamos usando la distancia mínima al borde, conseguiremos separar todos los objetos en la imagen.

- *Eliminación de ramas:* en algunos casos, la esqueletización genera algunas ramas que dificultan el reconocimiento del objeto, por lo que interesa eliminarlas. Para ello, se pasan una serie de elementos estructurales (8 en concreto) para reducir en un píxel su tamaño. El número de iteraciones del proceso depende de cada caso.

- *Localización de elementos:* a partir de una imagen dada I, en la que se encuentra el objeto A, de la imagen I^c y del elemento $B = X - A$ (siendo X una imagen ligeramente más grande que A toda ella a valor alto), realizamos las operaciones:

  - Erosionar I con A.
  - Erosionar I^c con X.
  - Obtener la intersección de las dos imágenes obtenidas.

  De esta forma detectamos el objeto A.

2.3. Redes neuronales

2.3.1. Introducción

Las redes neuronales (o redes de neuronas artificiales) son un paradigma de aprendizaje y procesamiento automático inspirado en la forma en que funciona el sistema nervioso de los animales. Se trata de un sistema de interconexión de neuronas en una red que colabora para producir un estímulo de salida. Forman parte de la disciplina científica conocida como Inteligencia Artificial. Podemos definir la neurona artificial como un elemento que posee un estado interno (nivel de activación) y recibe señales que le permiten, en su caso, cambiar de estado. El conjunto de estados de la neurona puede ser discreto o continuo. La función de activación o función de transición de estado es la encargada de permitir cambiar de estado a las neuronas en función de las señales que reciben. Estas señales pueden venir del exterior o de otras neuronas a las que está conectada. El nivel de activación de una célula depende de las entradas recibidas y de los valores sinápticos, pero no de los anteriores valores de estados internos. El siguiente modelo sigue el propuesto en [2, Apartado 1.2]. En primer lugar debemos calcular la entrada total a la célula E_i, que en se define como:

$$E = x_1 w_1 + x_2 w_2 + ... + x_n w_n \tag{2.55}$$

siendo $x_1, x_2, ..., x_n$ las entradas a la neurona artificial (vector $\bar{X}$) y $w_1, w_2, ..., w_n$ sus pesos asociados ($\bar{W}$). En forma vectorial:

$$E = X^T W \tag{2.56}$$

Esta señal E es procesada por la función de activación F, que produce la señal de salida S de la neurona. Existen diversas funciones de activación que veremos más adelante.

Estructura básica y funcionamiento de la red

A la manera en que las células se conectan entre sí se la denomina patrón de conectividad o arquitectura de la red. La estructura básica es la siguiente:

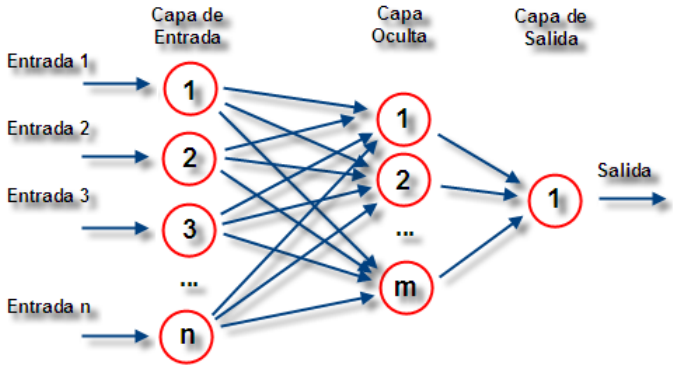

Figura 2.6: Red multicapa

Esta arquitectura es la típica para implementar el paradigma conocido como Retropropagación (Back Propagation), el empleado en este proyecto. Encontramos tres niveles: el primero constituido por las células de entrada, que reciben los valores del exterior; el segundo formado por una serie de capas ocultas; y por último el nivel de salida. Las células se comunican a través de las interconexiones, por donde reciben y envían valores numéricos, que son evaluados por los pesos de dichas conexiones. Estos pesos se ajustarán durante el entrenamiento de la red neuronal. Para cada vector de entrada, se copia el valor correspondiente a cada célula de entrada. Una vez recibidas todas las entradas, cada una de las células las procesa (mediante la función de activación) y genera una salida que se propaga a través de las interconexiones, llegando como entrada a una célula destino. Después de este proceso, se generará un vector de salida, con los valores de cada una de las células de salida.

Aprendizaje

El aprendizaje es la parte más importante de una red neuronal. Es donde definiremos el tipo de problemas que será capaz de resolver dicha red. Este aprendizaje se basa en

ejemplos, cuyo conjunto debe cumplir dos requisitos:

- Ser significativo. Debe haber un número suficiente de ejemplos. Si el conjunto es insuficiente, el cálculo de los pesos no será correcto.
- Ser representativo. El conjunto de aprendizaje debe estar formado por elementos diversos, de esta forma la red no se "especializará" en uno en concreto. Todas las regiones significativas del espacio de estados (valores posibles de las células) deben estar suficientemente representadas en el conjunto de aprendizaje.

En la fase de aprendizaje se calcularán los pesos sinápticos que ponderan las distintas interconexiones entre neuronas, de tal forma que la red sea capaz de resolver correctamente un problema concreto. El proceso general de aprendizaje consiste en introducir uno a uno todos los ejemplos del conjunto de aprendizaje y modificar los pesos de las conexiones según un esquema de aprendizaje. Una vez introducidos todos los pesos, se comprueba un criterio de convergencia; de no cumplirlo, se repite el proceso completo. Este criterio de convergencia depende del tipo de red o del tipo de problema a resolver. La finalización de un proceso de aprendizaje se puede determinar de las siguientes maneras:

- Número fijo de ciclos: se introduce el conjunto un número de veces decidido de antemano, dándose por aceptada la red resultante.
- Error por debajo de un valor dado: se define una función de error, y se finaliza el aprendizaje cuando dicho error alcanza un valor menor de un umbral dado. Existe la posibilidad de que el error no disminuya nunca hasta ese umbral, por lo que existe una condición de número de ciclos que finaliza el aprendizaje en el caso de que la red no converja.
- Modificación de pesos irrelevante: en algunos modelos las conexiones se modifican con menor intensidad en cada ciclo. Si el proceso continúa, llega un momento que los pesos no sufren variaciones, por lo que se considera que la red ha convergido y se finaliza el aprendizaje.

A continuación describiremos los distintos tipos de esquemas de aprendizaje.

Aprendizaje supervisado En este tipo de esquemas, los datos del conjunto de aprendizaje tienen dos tipos de atributos: por un lado los datos propiamente dichos, y por otro información relativa a la solución del problema. Cada vez que se introduce un ejemplo y se procesa para obtener una salida, dicha salida se compara con la salida que debería haber producido, y se modifican los pesos en relación a esa comparación.

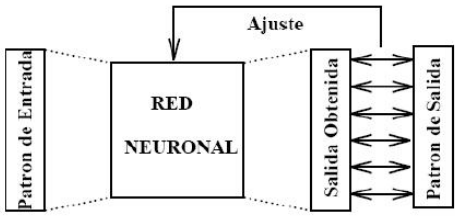

Figura 2.7: Esquema del aprendizaje supervisado

Aprendizaje no supervisado En este caso no se dispone de información sobre la solución del problema. La red modificará los pesos a partir de información interna únicamente (por este motivo también se les llama sistemas autoorganizados). Cuando se utiliza aprendizaje no supervisado, la red trata de determinar características de los datos del conjunto de entrenamiento como rasgos significativos, regularidades o redundancias.

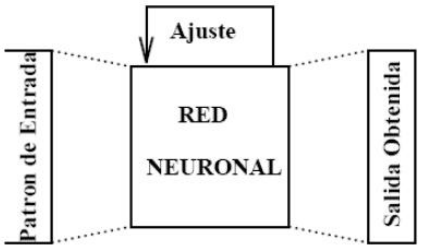

Figura 2.8: Esquema del aprendizaje no supervisado

Aprendizaje por refuerzo No se dispone de información concreta del error cometido por la red para cada ejemplo en el aprendizaje, sino que simplemente se determina si la salida producida para dicho ejemplo es o no correcta. Es por tanto una variante del aprendizaje supervisado. En este caso, aunque para todo el conjunto de aprendizaje consigamos obtener salidas adecuadas, no podemos asegurar la solución del problema, ya que desconocemos las salidas de los datos de entrada que se puedan presentar en el futuro.

Debemos tener presente que un ajuste muy bueno del conjunto de aprendizaje puede llevar a malos resultados cuando utilicemos la red con otros datos. Es lo que se denomina sobreajuste de los datos de entrenamiento. Para evitar este problema y poder determinar si la red produce salidas correctas con datos distintos de los del conjunto de aprendizaje, se divide dicho conjunto en dos: entrenamiento y validación. El primero servirá para calcular los pesos de las conexiones, y el segundo para medir el error cometido. De esta forma medimos la eficacia utilizando datos que no han sido

utilizados en el aprendizaje de la red. Si el error sobre el conjunto de validación es pequeño, podemos garantizar la capacidad de generalización de la red. Los conjuntos de entrenamiento y validación deben tener las siguientes características:

- El conjunto de validación debe ser independiente del de entrenamiento.
- El conjunto de validación debe cumplir los requisitos vistos anteriormente para el conjunto de aprendizaje.

También podemos utilizar el conjunto de validación durante el proceso de aprendizaje. En primer lugar calculamos los pesos de las conexiones con el conjunto de entrenamiento, según el esquema de aprendizaje que elijamos. Posteriormente introducimos el conjunto de validación y obtenemos el error cometido: si este error supera un umbral dado, volvemos a calcular los pesos con el conjunto de entrenamiento; si no, finalizamos el aprendizaje de la red.

2.3.2. Primeros modelos computacionales

Células de McCulloch-Pitts

Este primer ejemplo de red neuronal fue propuesto por Warren McCulloch y Walter Pitts en el artículo "A Logical Calculus of the Ideas Immanent in Nervous Activity", donde modelizaban una estructura y funcionamiento simplificado de las neuronas del cerebro, considerándolas como dispositivos con dos estados: apagado (0) y encendido (1).

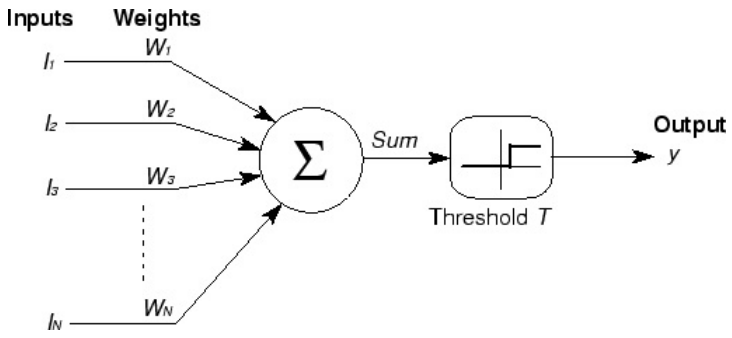

Figura 2.9: Esquema general de una neurona de McCulloch-Pitts

La salida viene definida por:

$$Y = \{ \begin{array}{ll} 1 & \text{si} \quad \sum I_i W_i > T \\ 0 & \text{en caso contrario} \end{array} \tag{2.57}$$

La célula de McCulloch-Pitts recibe como entrada un conjunto de N valores binarios (I_1, I_2,..., I_N) y genera una salida Y, también binaria. También dispone de $N+1$ valores reales: los N pesos de las conexiones (W_1, W_2, ..., W_N) y el umbral T, que puede ser distinto para cada célula. A partir de este esquema se define el primer modelo de red neuronal:

> Una red neuronal es una colección de neuronas de McCulloch y Pitts, todas ellas con las mismas escalas de tiempos, donde sus salidas están conectadas a las entradas de otras neuronas.

Una red neuronal de células de McCulloch-Pitts tiene la capacidad de computación universal, es decir, puede modelizar cualquier estructura que pueda ser programada en un computador. Los computadores están constituidos por elementos de cálculo simples; si modelizamos cada uno de estos elementos mediante una red de células de McCulloch-Pitts, podremos realizar los mismos cálculos con la correspondiente estructura de células.

Ejemplo: función lógica NOT Podemos considerarla como una célula de McCulloch-Pitts con una única entrada y una única salida, con un peso de valor -1 y un umbral de valor -1.

$$Y = \begin{cases} 1 & \text{si} \quad I * (-1) > -1 \\ 0 & \text{en caso contrario} \end{cases} \tag{2.58}$$

Por tanto, si $I = 0 \rightarrow Y = 1$; y si $I = 1 \rightarrow Y = 0$. Es decir, el comportamiento de la función lógica NOT.

Ejemplo: función lógica AND En este caso tenemos dos entradas y una salida; el valor de los pesos y del umbral es 1.

$$Y = \begin{cases} 1 & \text{si} \quad \sum I_i > 1 \\ 0 & \text{en caso contrario} \end{cases} \tag{2.59}$$

Por lo que tendremos:

$I_1 = I_2 = 0 \quad \rightarrow Y = 0$
$I_1 = 0; I_2 = 1 \quad \rightarrow Y = 0$
$I_1 = 1; I_2 = 0 \quad \rightarrow Y = 0$
$I_1 = I_2 = 1 \quad \rightarrow Y = 1$

Ejemplo: función lógica OR Podemos representarla por la misma célula de la función lógica AND, pero cambiando el valor del umbral por 0.

$$Y = \{ \begin{array}{ll} 1 & \text{si} \quad \sum I_i > 0 \\ 0 & \text{en caso contrario} \end{array} \tag{2.60}$$

Resultando:
$I_1 = I_2 = 0 \quad \rightarrow Y = 0$
$I_1 = 0; I_2 = 1 \quad \rightarrow Y = 1$
$I_1 = 1; I_2 = 0 \quad \rightarrow Y = 1$
$I_1 = I_2 = 1 \quad \rightarrow Y = 1$

En la práctica, con funciones más complejas o sistemas de computación sofisticados, es imposible utilizar células de McCulloch-Pitts, ya que el número de células y de parámetros involucrados sería muy elevado. Se necesitaría un mecanismo de asignación de parámetros automático, lo que se conoce como mecanismo de aprendizaje.

Perceptron

Este modelo se concibió como un sistema capaz de realizar tareas de clasificación de forma automática. La idea era que el sistema fuera capaz de determinar las ecuaciones de las superficies que hacían de frontera entre las distintas clases existentes. El sistema se basaba en la información de los ejemplos existentes de las distintas clases (patrones o ejemplos de entrenamiento). De esta forma podíamos determinar a qué clase pertenecía un ejemplo desconocido.

La arquitectura de esta red se basa en una estructura monocapa, en la que hay un conjunto de células de entrada y una o varias células de salida. El número de ellas depende del problema. Cada una de las células de entrada tiene conexiones con todas las células de salida; estas conexiones determinarán las superficies de discriminación del sistema.

En el ejemplo de la figura 2.10 tenemos dos entradas (x_1, x_2), una salida (y), los pesos de cada conexión (w_1, w_2) y un umbral (Θ). La salida se obtiene de la siguiente forma ([2, Apartado 2.2]):

Primero calculamos la activación de la célula de salida mediante suma ponderada de todas las entradas:

$$y' = \sum_{i=1}^{n} w_i x_i \tag{2.61}$$

La salida y se obtiene aplicando una función de salida al nivel de activación de la

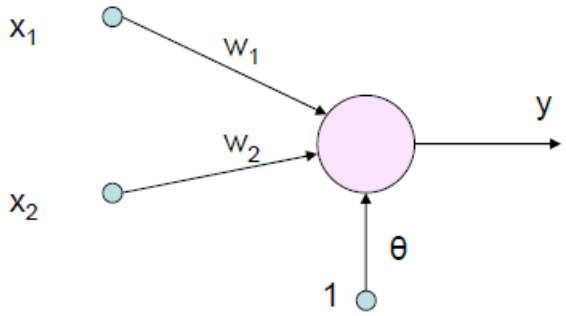

Figura 2.10: Arquitectura de un perceptrón simple con 2 entradas y una salida

célula. Esta función de salida es una función escalón de la forma:

$$y = F(y', \Theta) \tag{2.62}$$

$$F(s, \Theta) = \begin{cases} 1 & \text{si} \quad s > \Theta \\ -1 & \text{en caso contrario} \end{cases} \tag{2.63}$$

Pasando el término Θ al otro lado de la ecuación, tenemos:

$$y = F\left(\sum_{i=1}^{n} w_i x_i + \Theta\right) \tag{2.64}$$

donde F es:

$$F(s) = \begin{cases} 1 & \text{si} \quad s > 0 \\ -1 & \text{en caso contrario} \end{cases} \tag{2.65}$$

La función de salida F es binaria, por lo que podemos utilizarla fácilmente como discriminante de dos clases:

- Si la red produce salida 1 $\rightarrow$ la entrada pertenece a la clase A.
- Si la red produce salida -1 $\rightarrow$ la entrada pertenece a la clase B.

En un caso como el anterior, de dos dimensiones, la ecuación 2.64 se transforma en:

$$w_1 x_1 + w_2 x_2 + \Theta = 0 \tag{2.66}$$

una recta de pendiente $-\frac{w_1}{w_2}$ con ordenada en el origen $-\frac{\Theta}{w_1}$. Teniendo dos células de entrada, los patrones de entrenamiento pueden representarse como puntos en un espacio bidimensional. Si dichos puntos pertenecen a dos clases, la recta anterior representa la frontera entre ambos conjuntos de puntos. Para determinar la ecuación de la recta, es necesario el proceso de aprendizaje, donde asignaremos los valores correctos de los pesos para nuestro problema.

Este proceso de entrenamiento se realiza de la siguiente forma (ejemplo con dos clases, A y B). Introducimos un patrón de entrenamiento de la clase A. Calculamos la salida que genera la red para dicho patrón, utilizando pesos y umbral aleatorios. Si la respuesta de la red es correcta (tomamos salida igual a 1 para la clase A), no se modifica nada y se introduce otro ejemplo. Si la salida es incorrecta, se modifican los pesos:

$$\Delta w_i = d(\chi)x_i \tag{2.67}$$

donde w_i es el peso asociado a la conexión de la entrada x_i con la salida y, y $d(\chi)$ es la salida correcta (en este caso, $d(\chi) = 1$).
El umbral también debe ser modificado: $\Delta\Theta = d(\chi)$, ya que podemos considerar el umbral como el peso de una entrada con valor constante e igual a 1 ($x_0 = 1$). Por tanto, podemos agruparlo en:

$$\Delta w_i = d(\chi)x_i \qquad i = 0, ..., n$$

Esta regla es fundamentalmente una regla de aprendizaje por refuerzo en la que se potencian las salidas correctas y no se tienen en cuenta las incorrectas. No existe ninguna graduación en la regla que indique cómo de errónea es la salida que produce nuestra red y refuerce en función de ese error.

Adaline

El perceptrón es una sistema de aprendizaje con salidas binarias, por lo que sólo puede codificar un conjunto discreto de estados. Por este motivo es capaz de realizar tareas de clasificación, pero éstas no son los únicos problemas abordables desde la perspectiva del aprendizaje. Si las salidas fueran números reales, podrían codificar cualquier tipo de salida, siendo sistemas de resolución de problemas generales. En ese caso, podrían resolver problemas a partir de ejemplos en los que es necesario aproximar una función cualquiera $F(x)$, definida por un conjunto de datos de entrada. Como podemos ver en [2, Apartado 2.3], los datos de entrenamiento son:

$$P = (\overrightarrow{x_1}, y_1), ..., (\overrightarrow{x_m}, y_m)$$

donde $\overrightarrow{x_i}$ es el vector de entrada e y_i es su salida asociada.

La función a aproximar sería:

$$F(\overrightarrow{x_i}) = y_i, \quad \forall p \in P$$

Es decir, habrá que obtener una función $F(\overrightarrow{x})$ tal que aplicada a cada una de las entradas $\overrightarrow{x_i}$ del conjunto de aprendizaje P produzca la salida y_i correspondiente.

En 1960, Widrow y Hoff propusieron un sistema de aprendizaje que sí tuviera en cuenta el error producido, y diseñaron lo que denominaron Adaptative Linear Neuron, Adaline. La estructura es casi idéntica a la del perceptrón. Es un elemento combinador adaptativo, es decir, recibe un conjunto de entradas y las combina para producir una salida. Esta salida puede transformarse en binaria mediante un conmutador bipolar (1 si la salida es positiva y -1 si la salida es negativa).

El aprendizaje en este caso sí tiene en cuenta la diferencia entre el valor que produce la capa de salida y^p para un patrón de entrada x^p y el valor que debería haber producido d^p ($|d^p - y^p|$). A esta regla de aprendizaje se la conoce como *regla Delta*. Esta regla utiliza, a diferencia de la regla de aprendizaje del perceptrón, la salida de la red directamente, sin pasarla por ninguna función umbral. El objetivo es obtener una salida $y = d^p$ al introducir el vector de entrada x^p. De nuevo el problema será encontrar el valor de los pesos w_i. No es posible encontrar una salida exacta, pero sí minimizar el error cometido por la red para todos los patrones. Por este motivo, necesitamos una medida de error global, como el error cuadrático medio (aunque otros pueden ser utilizados).

$$E = \sum_{p=1}^{m} E^p = \frac{1}{2} \sum_{p=1}^{m} (d^p - y^p)^2 \tag{2.68}$$

La regla intentará minimizar este valor para todos los elementos del conjunto de aprendizaje, por ser una medida de error global. La manera de minimizar este error es recurrir a un proceso iterativo en el que se modifican los pesos de las conexiones mediante la *Regla del descenso del gradiente*.

La idea es modificar cada peso proporcionalmente a la derivada del error respecto del peso, evaluada en el patrón actual:

$$\Delta_p w_j = -\gamma \frac{\partial E^p}{\partial w_j} \tag{2.69}$$

Aplicando la regla de la cadena, tenemos:

$$\frac{\partial E^p}{\partial w_j} = \frac{\partial E^p}{\partial y^p} \frac{\partial y^p}{\partial w_j} \tag{2.70}$$

Al ser unidades lineales, sin función de activación en la salida, se cumple:

$$\frac{\partial y^p}{\partial w_j} = x_j \qquad \frac{\partial E^p}{\partial y^p} = -(d^p - y^p) \tag{2.71}$$

Sustituyendo, finalmente obtenemos:

$$\Delta_p w_j = \gamma (d^p - y^p) x_j \tag{2.72}$$

A partir de la expresión de la Ecuación 2.85 podemos llegar a la regla del perceptrón. Si aplicamos a la salida del Adaline el conmutador bipolar comentado anteriormente y tomamos $\gamma = 1$, tenemos:

$$\Delta_p w_j = \begin{cases} \gamma x_j & d^p > y^p \\ -\gamma x_j & \text{si } d^p < y^p \\ 0 & \text{en caso contrario} \end{cases}$$

Por tanto, podemos considerar la regla Delta como una extensión de la regla del perceptrón a valores de salida reales.

El procedimiento de aprendizaje de la regla Delta será:
1. Inicializar los pesos aleatoriamente.
2. Introducir un patrón de entrada.
3. Calcular la salida de la red y^p, compararla con la deseada d^p y obtener la diferencia $(d^p - y^p)$.
4. Para todos los pesos, hacer $(d^p - y^p)x_i$ y ponderarla por una tasa de aprendizaje γ.
5. Modificar el peso, restando del valor antiguo la cantidad obtenida en 4.
6. Si no se ha alcanzado el criterio de convergencia, volver a 2; si se han acabado todos los patrones, empezar de nuevo a introducirlos.

Diferencias entre modelos perceptrón y Adaline

- En el perceptrón la salida es binaria, mientras que en el Adaline es real.
- En el perceptrón, si entrada y salida pertenecen a la misma categoría su diferencia es 0; en caso contrario es ± 1. En el Adaline se calcula la diferencia real entre entrada y salida.
- En el Adaline se conoce el error que comete la red; en el perceptrón sólo se determina si se ha equivocado o no.
- En el Adaline hay una tasa de aprendizaje γ para regular la influencia de cada equivocación en la modificación de los pesos.

2.3.3. Perceptrón multicapa

El perceptrón multicapa es una generalización del perceptrón simple. Minsky y Papert (1969) mostraron que la combinación de varios perceptrones simples (inclusión de neuronas ocultas) podía ser una solución para ciertos problemas no lineales, pero no indicaron cómo calcular los pesos asociados a cada conexión. Sí lo hicieron Rumelhart,

Hinton y Williams (1986) con la *regla Delta generalizada*, una manera de retropropagación de los errores medidos en la salida de la red hacia las neuronas ocultas.

Otros autores como Cybenko (1989) o Hornik (1989) han demostrado que cualquier función continua sobre un compacto de $\Re^n$ puede aproximarse con un perceptrón multicapa con, al menos, una capa oculta de neuronas.

Es un modelo muy adecuado para abordar problemas reales, ya que es capaz de aprender a partir de un conjunto de ejemplos, aproximar relaciones no lineales, filtrar ruidos en los datos, etc., pero no significa que sea el mejor aproximador universal. Dentro de las redes de neuronas, el perceptrón multicapa es una de las arquitecturas más utilizadas por su capacidad de aproximador universal ya comentada y por su fácil uso y aplicabilidad. A pesar de ello, el perceptrón multicapa tiene una serie de limitaciones como el largo proceso de aprendizaje para problemas complejos con gran número de variables; la dificultad para codificar problemas reales mediante valores numéricos; la dificultad para realizar un análisis teórico de la red debido a la presencia de componentes no lineales y a la alta conectividad. A continuación analizaremos dicho modelo, tomando como referencia [2, Cap. 3].

Arquitectura

Se caracteriza por tener sus neuronas agrupadas en capas de diferentes niveles. Existen tres tipos de capas: de entrada, ocultas y de salida.

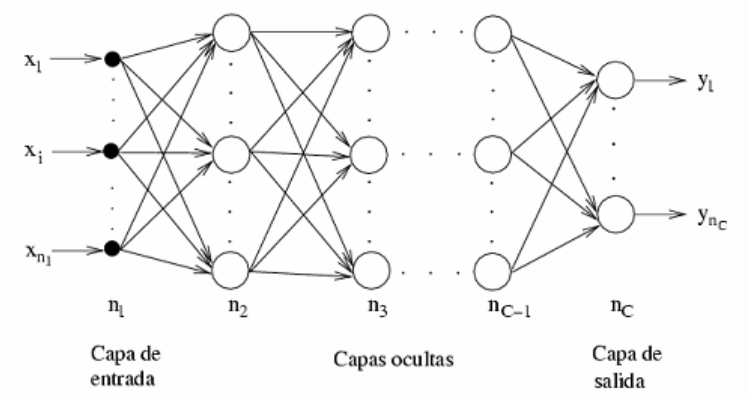

Figura 2.11: Arquitectura del perceptrón multicapa

Las neuronas de la capa de entrada se encargan únicamente de recibir señales o patrones del exterior y propagarlas a todas las neuronas de la siguiente capa. La última capa actúa como salida de la red, proporcionando al exterior la respuesta de la misma para cada uno de los patrones de entrada. En las capas ocultas se realiza un procesamiento no lineal de los patrones recibidos. En la Figura 2.11 observamos que

todas las conexiones se realizan hacia adelante; por este motivo se las denomina redes alimentadas hacia adelante o feedforward. Como en las arquitecturas ya vistas, cada conexión tiene un peso asociado, que en el perceptrón multicapa es un número real. Igual que en el modelo del perceptrón simple, existe un umbral en cada neurona que podemos tratar como una entrada de valor constante e igual a 1 con un peso determinado (el valor del umbral). Generalmente existe conectividad total, es decir, cada una de las neuronas de una capa está conectada con todas las neuronas de la capa siguiente, aunque también podemos considerar como perceptrón multicapa casos en los que esto no ocurra (conexiones con neuronas de capas no inmediatamente posteriores o ausencia de ciertas conexiones). La mayoría de las veces se parte de una red totalmente conectada y se eliminan ciertas conexiones según la naturaleza del problema, aunque no podemos asegurar que se produzcan mejores resultados.

Propagación de los patrones de entrada

Sea un perceptrón multicapa con C capas ($C-2$ capas ocultas) y n_c neuronas en la capa c, para $c = 1, 2, ..., C$. Sea $W^c = (w_{ij}^c)$ la matriz de pesos asociada a las conexiones de la capa c con la capa $c+1$ para $c = 1, 2, ..., C-1$, donde w_{ij}^c representa el peso de la conexión de la neurona i de la capa c con la neurona j de la capa $c+1$; y sea $U^c = (u_i^c)$ el vector de umbrales de las neuronas de la capa c, para $c = 2, 3, ..., C$. Se denota a_i^c a la activación de la neurona i de la capa c. Estas activaciones se calculan de la siguiente manera:

- Activación de las neuronas de la capa de entrada (a_i^1): la capa de entrada se encarga de transmitir hacia la red las señales recibidas desde el exterior:

$$a_i^1 = x_i \quad \text{para} \quad i = 1, 2, ..., n_1 \tag{2.73}$$

  siendo $X = (x_1, x_2, ..., x_{n1})$ el patrón de entrada a la red.

- Activación de las neuronas de la capa oculta (a_i^c): en las capas ocultas se procesa la información recibida aplicando la función de activación f a la suma de los productos de las activaciones que recibe por sus correspondientes pesos:

$$a_i^c = f\left(\sum_{j=1}^{n_c-1} w_{ji}^{c-1} a_j^{c-1} + u_i^c\right) \quad \text{para} \quad i = 1, 2, ..., n_c \quad \text{y} \quad c = 2, 3, ..., C-1 \tag{2.74}$$

  siendo a_j^{c-1} las activaciones de las neuronas de la capa $c-1$.

- Activación de las neuronas de la capa de salida (a_i^C): análogamente al caso de las capas ocultas:

$$y_i = a_i^C = f\left(\sum_{j=1}^{n_C-1} w_{ji}^{C-1} a_j^{C-1} + u_i^C\right) \quad \text{para} \quad i = 1, 2, ..., n_C \tag{2.75}$$

siendo $Y = (y_1, y_2, ..., y_{n_C})$ el vector de salida de la red.

La función f empleada es la denominada *función de activación*. En esta arquitectura, las más empleadas son:

- Función sigmoidal:

$$f_1(x) = \frac{1}{1 + e^{-x}} \tag{2.76}$$

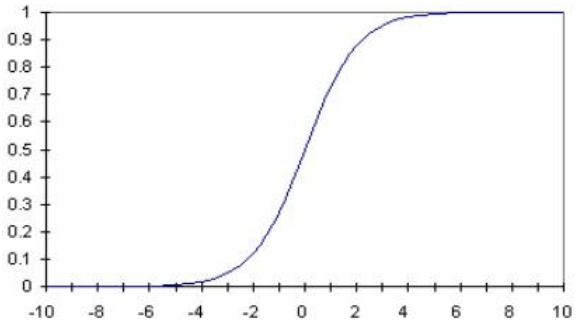

Figura 2.12: Función sigmoidal

- Función tangente hiperbólica:

$$f_2(x) = \frac{1 - e^{-x}}{1 + e^{-x}} \tag{2.77}$$

Dichas funciones están acotadas en el intervalo [0, 1] y [-1, 1] respectivamente; son funciones crecientes con un nivel de saturación máximo de 1 y mínimo de 0 en la sigmoidal y -1 en la tangente hiperbólica.

La elección de una función de activación u otra depende del diseñador de la red, que decidirá según los valores de activación que desee que alcancen las neuronas. Esta elección no influye generalmente en la capacidad de la red para resolver un problema. Normalmente todas las neuronas de la red utilizan la misma función de activación,

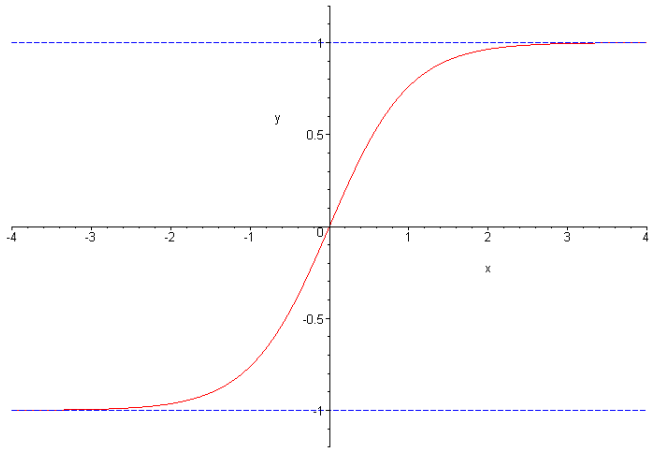

Figura 2.13: Función tangente hiperbólica

aunque en algunos casos las neuronas de la capa de salida pueden utilizar otra diferente, como la función identidad o la función escalón, según la naturaleza del problema.

En general, podemos escribir que:

$$Y = F(X, W) \tag{2.78}$$

donde Y es el vector de salida de la red; X es el vector de entrada a la red; W es la matriz de parámetros de la red (pesos de las conexiones y umbrales de las neuronas); y F es una función continua no lineal.

Diseño de la arquitectura del perceptrón multicapa

Existen varios parámetros que el diseñador debe definir a la hora de crear una red: la función de activación a emplear, el número de neuronas y el número de capas en la red.

Como hemos comentado anteriormente, la elección de la función de activación depende de los valores de activación que busquemos y no influye en la eficacia de nuestra red a la hora de resolver el problema.

Para determinar el número de neuronas y de capas, debemos tener en cuenta que el número de neuronas de las capas de entrada y salida nos viene impuesto por las variables de nuestro problema, aunque en primer lugar debemos decidir cuáles son relevantes y cuáles no; de lo contrario podemos complicar en gran medida el aprendizaje de la red.

Para las capas ocultas, el diseñador debe elegir su número y el número de neuronas en ellas. No existe un método para esta elección y en la mayoría de los casos se determina por ensayo y error, por lo que no podemos asegurar que hayamos alcanzado la solución óptima. El número de neuronas ocultas no es un parámetro significativo, ya que podremos resolver un problema de manera adecuada con distintas configuraciones,

aunque sí puede influir en la capacidad de generalización de la red. Actualmente existen líneas de investigación abiertas centradas en la determinación automática del número óptimo de capas y neuronas ocultas, basadas en técnicas evolutivas.

Algoritmo de Retropropagación

La regla o algoritmo de aprendizaje es el mecanismo mediante el cual se van calculando los parámetros de la red. En el perceptrón multicapa se trata de un algoritmo de aprendizaje supervisado, se modifican los parámetros para minimizar el error entre la salida producida por la red y la salida deseada, es decir, $Min_W E$, siendo E la función error y W el conjunto de parámetros de la red. En general:

$$E = \frac{1}{N}\sum_{n=1}^{N} e(n) \tag{2.79}$$

donde N es el número de patrones y $e(n)$ es el error cometido por la red para el patrón n. Utilizando el error cuadrático medio, tenemos:

$$e(n) = \frac{1}{2}\sum_{i=1}^{n_C}(s_i(n) - y_i(n))^2 \tag{2.80}$$

siendo $Y(n) = (y_i(n), ..., y_{n_C}(n))$ y $S(n) = (s_i(n), ..., s_{n_C}(n))$ los vectores de salida de la red y salida deseada para el patrón n, respectivamente.

El aprendizaje del perceptrón multicapa consiste en minimizar una función error. Para resolver ese problema son necesarias técnicas de optimización no lineales, ya que estamos trabajando con funciones de activación también no lineales. Por tanto la respuesta de la red no será lineal respecto a los parámetros ajustables. Esas técnicas se basan generalmente en una adaptación de los parámetros siguiendo una cierta dirección de búsqueda (*método del descenso del gradiente*).

El procedimiento más utilizado está basado en métodos del gradiente estocástico, una sucesiva minimización de los errores para cada patrón $e(n)$. Aplicando el método de descenso del gradiente estocástico, tenemos:

$$w(n) = w(n-1) - \alpha\frac{\partial e(n)}{\partial w} \tag{2.81}$$

donde w es cada parámetro de la red, $e(n)$ es el error para el patrón n y α es la razón o tasa de aprendizaje.

El algoritmo de retropropagación consiste en aplicar este método de descenso del gradiente de forma eficiente en el perceptrón multicapa: el error cometido en la salida de la red se propaga hacia atrás, transformándolo en el error de cada una de las neuronas ocultas de la red.

Regla Delta generalizada

Para el desarrollo de esta regla distinguimos dos casos: uno para los pesos de la capa oculta $C-1$ a la capa de salida y los umbrales de las neuronas de salida (*Caso 1*); y otro para el resto de pesos y umbrales de la red (*Caso 2*).

Caso 1 Sea w_{ji}^{C-1} el peso de la conexión de la neurona j de la neurona $C-1$ con la neurona i de la capa de salida. Según la Ecuación 2.81, tenemos:

$$w_{ji}^{C-1}(n) = w_{ji}^{C-1}(n-1) - \alpha \frac{\partial e(n)}{\partial w_{ji}^{C-1}} \tag{2.82}$$

Por tanto es necesario evaluar la derivada del error $e(n)$ en dicho punto para actualizar el peso w_{ji}^{C-1}. Según la Ecuación 2.80 y teniendo en cuenta que la salida deseada para la red $S(n)$ no depende del peso y que w_{ji}^{C-1} sólo afecta a la neurona de salida i, tenemos:

$$\frac{\partial e(n)}{\partial w_{ji}^{C-1}} = -\left(s_i(n) - y_i(n)\right) \frac{\partial y_i(n)}{\partial w_{ji}^{C-1}} \tag{2.83}$$

Para calcular $\dfrac{\partial y_i(n)}{\partial w_{ji}^{C-1}}$, nos servimos de la Ecuación 2.75. Aplicando la regla de la cadena y teniendo en cuenta que el único término del sumatorio en el que interviene el peso w_{ji}^{C-1} es $w_{ji}^{C-1}a_j^{C-1}$, obtenemos:

$$\frac{\partial y_i(n)}{\partial w_{ji}^{C-1}} = f'\left(\sum_{j=1}^{n_C-1} w_{ji}^{C-1}a_j^{C-1} + u_i^C\right) a_j^{C-1}(n) \tag{2.84}$$

Podemos definir el término $\delta_i^C(n)$ asociado a la neurona i de la capa de salida y al patrón n como:

$$\delta_i^C(n) = -(s_i(n) - y_i(n)) f'\left(\sum_{j=1}^{n_C-1} w_{ji}^{C-1}a_j^{C-1} + u_i^C\right) \tag{2.85}$$

Sustituyendo la Ecuación 2.84 en 2.83 y utilizando la Ecuación 2.85, llegamos a:

$$\frac{\partial e(n)}{\partial w_{ji}^{C-1}} = \delta_i^C(n) a_j^{C-1}(n) \tag{2.86}$$

$$w_{ji}^{C-1}(n) = w_{ji}^{C-1}(n-1) - \alpha\delta_i^C(n)a_j^{C-1}(n) \quad \text{para} \quad j = 1, 2, ..., n_{C-1} \quad i = 1, 2, ...n_C \tag{2.87}$$

En esta última ecuación se expresa la ley para modificar el peso w_{ji}^{C-1} según la activación de la neurona de origen de la conexión (neurona j de la capa $C-1$) y el término δ de la neurona de destino (neurona de salida i), término que contiene el error cometido por la red para dicha neurona de salida.

Como se ha dicho anteriormente, los umbrales se consideran entradas constantes e iguales a 1, cuyo peso nos da el valor del umbral. De tal forma, los umbrales de las neuronas de la capa de salida se modifican según la expresión:

$$u_i^C(n) = u_i^C(n-1) + \alpha\delta_i^C(n) \quad \text{para} \quad i = 1, 2, ..., n_C \tag{2.88}$$

Caso 2 Tomamos un peso de la capa $C-2$ a la capa $C-1$. Sea w_{kj}^{C-2} el peso de la conexión de la neurona k de la capa $C-2$ a la neurona j de la capa $C-1$. Análogamente al *Caso 1*, tenemos:

$$w_{kj}^{C-2}(n) = w_{kj}^{C-2}(n-1) - \alpha\frac{\partial e(n)}{\partial w_{kj}^{C-2}} \tag{2.89}$$

En este caso, el peso w_{kj}^{C-2} influye en todas las salidas de la red, por lo que la derivada del error $e(n)$ viene dada por la suma de las derivadas para cada una de las salidas:

$$\frac{\partial e(n)}{\partial w_{kj}^{C-2}} = -\sum_{i=1}^{n_C}\left(s_i(n) - y_i(n)\right)\frac{\partial y_i(n)}{\partial w_{kj}^{C-2}} \tag{2.90}$$

En este caso, para calcular $\dfrac{\partial y_i(n)}{\partial w_{kj}^{C-2}}$ debemos tener en cuenta que este peso influye en la activación de la neurona j de la capa $C-1$ (a_j^{C-1}) y que el resto de las activaciones de las neuronas de la capa $C-1$ no depende de dicho peso, obteniendo:

$$\frac{\partial y_i(n)}{\partial w_{kj}^{C-2}} = f'\left(\sum_{j=1}^{n_{C-1}} w_{ji}^{C-1}a_j^{C-1} + u_i^C\right) w_{ji}^{C-1}\frac{\partial a_j^{C-1}}{\partial w_{kj}^{C-2}}(n) \tag{2.91}$$

Sustituyendo la Ecuación 2.91 en 2.90 y utilizando la Ecuación 2.85 vista en el *Caso 1*, tenemos:

$$\frac{\partial e(n)}{\partial w_{kj}^{C-2}} = \sum_{i=1}^{n_C}\delta_i^C(n)w_{ji}^{C-1}\frac{\partial a_j^{C-1}}{\partial w_{kj}^{C-2}} \tag{2.92}$$

Por último debemos calcular $\dfrac{\partial a_j^{C-1}}{\partial w_{kj}^{C-2}}$ para obtener la ley de aprendizaje para el peso w_{kj}^{C-2}. Aplicando la regla de la cadena en la Ecuación 2.74:

$$\frac{\partial a_j^{C-1}}{\partial w_{kj}^{C-2}} = f'\left(\sum_{k=1}^{n_{C-2}} w_{kj}^{C-2}a_k^{C-2} + u_k^{C-1}\right) a_k^{C-2}(n) \tag{2.93}$$

Definimos $\delta_j^{C-1}(n)$ como:

$$\delta_j^{C-1}(n) = f'\left(\sum_{k=1}^{n_{C-2}} w_{kj}^{C-2}a_k^{C-2} + u_j^{C-1}\right)\sum_{i=1}^{n_C}\delta_i^C(n)w_{ji}^{C-1} \tag{2.94}$$

Sustituyendo la Ecuación 2.93 en 2.92 y teniendo en cuenta la definición de $\delta_j^{C-1}(n)$ vista en la Ecuación 2.94, obtenemos:

$$\frac{\partial e(n)}{\partial w_{kj}^{C-2}} = \delta_j^{C-1}(n) a_k^{C-2}(n) \tag{2.95}$$

Finalmente, podemos expresar la ley de aprendizaje para modificar el peso w_{kj}^{C-2} como:

$$w_{kj}^{C-2}(n) = w_{kj}^{C-2}(n-1) + \alpha \delta_j^{C-1}(n) a_k^{C-2}(n) \quad \text{para} \quad k = 1, 2, ..., n_{C-2} \quad \text{y} \quad j = 1, 2, ..., n_{C-1} \tag{2.96}$$

Como ocurre en el *Caso 1*, para modificar el peso de la conexión de la neurona k de la capa $C-2$ a la neurona j de la capa $C-1$, w_{kj}^{C-2}, basta considerar la activación de la neurona de origen (neurona k de la capa $C-2$) y el término δ de la neurona destino (neurona j de la capa $C-1$). La diferencia radica en el término δ, cuya expresión varía respecto al *Caso 1* y viene dado por la derivada de la función de activación y por la suma de los términos δ asociados a las neuronas de la siguiente capa (Ecuación 2.94). Generalizando la Ecuación 2.96 para una capa cualquiera c, $c = 1, 2, ..., C-2$, tenemos:

$$w_{kj}^{c}(n) = w_{kj}^{c}(n-1) + \alpha \delta_j^{c+1}(n) a_k^{c}(n)$$
$$\text{para} \quad k = 1, 2, ..., n_c, \quad j = 1, 2, ..., n_{c+1} \quad \text{y} \quad c = 1, 2, ..., C-2 \tag{2.97}$$

En este *Caso 2* volveremos a considerar los umbrales como entradas de valor constante e igual a 1, resultando la ley para modificarlos:

$$u_j^{c+1}(n) = u_j^{c+1}(n-1) + \alpha \delta_j^{c+1}(n) \quad \text{para} \quad j = 1, 2, ..., n_{c+1} \quad \text{y} \quad c = 1, 2, ..., C-2 \tag{2.98}$$

Resumen algoritmo retropropagación

Después de explicar en detalle el algoritmo y sus expresiones, vamos a intentar resumirlo en pocas palabras. Cada neurona de salida distribuye hacia atrás su error (valor δ) a todas las neuronas ocultas que se conectan a ella, ponderado por el peso de la conexión. La suma de estas cantidades es el término δ asociado a la neurona oculta. Este valor permite obtener los términos δ de las neuronas de la capa anterior y así sucesivamente hasta llegar a la primera capa oculta, de ahí el nombre de retropropagación.

A continuación enumeramos las expresiones finales que hemos hallado, considerando una función sigmoidal como función de activación:

Pesos de la capa oculta $C-1$ a la capa de salida y umbrales de la capa de salida

- Pesos

$$w_{ji}^{C-1}(n) = w_{ji}^{C-1}(n-1) - \alpha\delta_i^C(n)a_j^{C-1}(n) \quad \text{para} \quad j = 1, 2, ..., n_{C-1} \quad i = 1, 2, ...n_C \tag{2.99}$$

- Umbrales

$$u_i^C(n) = u_i^C(n-1) + \alpha\delta_i^C(n) \quad \text{para} \quad i = 1, 2, ..., n_C \tag{2.100}$$

- Delta

$$\delta_i^C(n) = -(s_i(n) - y_i(n))y_i(n)(1 - y_i(n)) \tag{2.101}$$

Pesos de la capa c a la capa $c+1$ y umbrales de las neuronas de la capa $c+1$ para $c = 1, 2, ..., C+2$

- Pesos

$$w_{kj}^c(n) = w_{kj}^c(n-1) + \alpha\delta_j^{c+1}(n)a_k^c(n)$$
$$\text{para} \quad k = 1, 2, ..., n_c, \quad j = 1, 2, ..., n_{c+1} \quad \text{y} \quad c = 1, 2, ..., C-2 \tag{2.102}$$

- Umbrales

$$u_j^{c+1}(n) = u_j^{c+1}(n-1) + \alpha\delta_j^{c+1}(n) \quad \text{para} \quad j = 1, 2, ..., n_{c+1} \quad \text{y} \quad c = 1, 2, ..., C-2 \tag{2.103}$$

- Delta

$$\delta_j^{c+1}(n) = a_j^c(n)(1 - a_j^c(n)) \sum_{i=1}^{n_{c+1}} \delta_i^{c+2}(n)w_{ji}^c \tag{2.104}$$

Razón de aprendizaje

En las Ecuaciones 2.99 y 2.96 aparece un parámetro α, denominado razón o tasa de aprendizaje. Este factor controla cuánto se desplazan los pesos de la red en la superficie del error siguiendo la dirección negativa del gradiente. Valores altos de α pueden provocar que nos "saltemos" un mínimo o incluso que el método oscile alrededor de él, mientras que valores demasiado pequeños ralentizan la convergencia del algoritmo. Para evitar inestabilidad por la tasa de aprendizaje, incluimos un segundo término en la ley de aprendizaje, llamado momento:

$$w(n) = w(n-1) - \alpha\frac{\partial e(n)}{\partial w} + \eta\Delta w(n-1) \tag{2.105}$$

siendo $\Delta w(n-1) = w(n-1) - w(n-2)$, es decir, el incremento que sufrió el parámetro w en la iteración anterior, y η un número positivo que controla la importancia asignada a ese incremento.

Esta regla fue propuesta por Rumelhart (1986) y no altera las propiedades de las expresiones vistas anteriormente, únicamente incorpora en el método cierta inercia haciendo que la modificación actual del parámetro dependa de la dirección de la anterior. Podemos escribir la ley de aprendizaje como:

$$w(n) = w(n-1) - \alpha \sum_{t=0}^{n} \eta^{n-t} \frac{\partial e(t)}{\partial w} \tag{2.106}$$

Observamos que el cambio actual de un parámetro viene dado por la suma de los gradientes del error para todas las iteraciones anteriores. De esta forma, si $\frac{\partial e(t)}{\partial w}$ tiene signos opuestos en iteraciones sucesivas, la suma contrarrestará ese cambio de signo y el cambio en el parámetro será mas suave; también, si $\frac{\partial e(t)}{\partial w}$ tiene el mismo signo en iteraciones sucesivas, la modificación del parámetro será mayor, acelerando la convergencia del algoritmo.

Aprendizaje del perceptrón multicapa A continuación enumeraremos los pasos que componen el proceso de aprendizaje del perceptrón multicapa. Es importante decir que generalmente los patrones de entrada y salida se encuentran normalizados mediante una transformación lineal a los intervalos [0, 1] y [-1, 1], según trabajemos con una función de activación sigmoidal o tangente hiperbólica. El aprendizaje también se puede realizar sin esa normalización, pero teniendo en cuenta que si los patrones de salida no están escalados la función de activación para la capa de salida debe ser la identidad.

- Paso 1: se inicializan aleatoriamente los pesos y umbrales de la red.
- Paso 2: se toma un patrón n del conjunto de entrenamiento $(X(n), S(n))$ y se propaga hacia la salida de la red (utilizando las Ecuaciones 2.73, 2.74 y 2.75) produciendo la salida de la red, $Y(n)$.
- Paso 3: se evalúa el error cuadrático medio cometido por la red para el patrón n (Ecuación 2.80).
- Paso 4: se aplica la regla Delta generalizada:
  - Se calculan los valores δ para todas las neuronas de la capa de salida (Ecuación 2.101).
  - Se calculan los valores δ para el resto de neuronas (Ecuación 2.104.

- Se modifican pesos y umbrales (Ecuaciones 2.99, 2.100, 2.96, 2.103).

- Paso 5: se repiten los pasos 2, 3 y 4 para todos los patrones de entrenamiento, completando así una iteración o ciclo de aprendizaje.

- Paso 6: se evalúa el error total E (error de entrenamiento) cometido por la red (Ecuación 2.79).

- Paso 7: se repiten los pasos 2, 3, 4, 5, y 6 hasta alcanzar un mínimo del error de entrenamiento, para lo cual se realizan m ciclos de aprendizaje.

Como ya vimos en el apartado referente al aprendizaje de redes neuronales, éste puede darse por finalizado cuando el error $\frac{\partial E}{\partial w} \approx 0$ o cuando se alcance un número determinado de ciclos.

Capacidad de generalización

Una vez entrenada nuestra red neuronal, debemos evaluar su comportamiento. No sólo es importante que la red haya aprendido correctamente los patrones de entrenamiento, sino que debemos conocer su funcionamiento ante patrones desconocidos, ya que éste es el verdadero propósito de la red. Durante el aprendizaje la red debe extraer las características de las muestras, para luego poder responder correctamente ante patrones diferentes. Es lo que se denomina capacidad de generalización de la red.

Para comprobar si una red tiene buena capacidad de generalización, dispondremos de dos conjuntos de patrones: el *conjunto de entrenamiento*, que utilizamos durante el proceso de aprendizaje; y el *conjunto de validación*, que nos servirá para medir los resultados de nuestra red. Estos conjuntos se obtienen de la totalidad de las muestras disponibles, y es conveniente que se separen de forma aleatoria.

No es recomendable un entrenamiento exhaustivo de la red, ya que penaliza su capacidad de generalización. Por este motivo, es recomendable presentar a la red el conjunto de patrones de validación pasado un número de ciclos para conocer el error cometido sobre dicho conjunto. Podemos encontrar dos situaciones: que los errores de entrenamiento y de validación permanezcan estables después de un número determinado de ciclos, o que el error de validación aumente a partir de un número de ciclos, manteniéndose el de entrenamiento constante. El primer caso es el deseado, puesto que significa que el aprendizaje ha acabado con éxito y la red tiene buena capacidad de generalización. El segundo implica un sobreentrenamiento: hemos conseguido una red que funciona muy bien con los patrones de entrenamiento, pero con patrones distintos el

resultado no es el esperado. En este caso debemos finalizar el aprendizaje en el momento en el que el error de validación alcanza su mínimo, aunque el error de entrenamiento se pueda mejorar, ya que conseguiremos cierta capacidad de generalización.

Deficiencias del algoritmo de retropropagación

A pesar de su buen resultado a la hora de entrenar el perceptrón multicapa, este algoritmo presenta dos deficiencias principales:

- Mínimos locales
  El proceso de adaptación de parámetros (pesos y umbrales) finaliza cuando $\frac{\partial E}{\partial w} \approx 0$, lo cual no implica necesariamente que se haya alcanzado un mínimo global. La superficie definida por el error E es compleja y llena de valles y colinas, por lo que lo que podemos caer en el error de considerar un mínimo local como global y finalizar el aprendizaje antes de tiempo.

  Una posible forma de evitar este problema es aumentar el número de neuronas ocultas en la red, ya que aumentamos el número de parámetros libres y el poder de representación interna de la red. Otras maneras de evitarlo son utilizar una tasa de aprendizaje decreciente a lo largo del proceso, partir de otras inicializaciones de los parámetros de la red o añadir ruido al método del descenso del gradiente.

- Parálisis
  Se produce cuando la entrada total a una neurona de la red toma valores muy grandes (tanto positivos como negativos). Debido a que las funciones de activación están acotadas (ver Figuras 2.12 y 2.13), si la entrada a una neurona tiene un valor muy alto, ésta se satura y alcanza un valor de activación máximo o mínimo. Cuando ocurre el fenómeno de parálisis en un perceptrón multicapa, los parámetros de la red permanecen invariables, por lo que la suma de errores locales permanece constante por un período largo de tiempo, pudiendo volver a decrecer después. Para evitar este problema, es recomendable inicializar aleatoriamente los parámetros de la red con valores próximos a cero.

Capítulo 3

Implementación

En este capítulo se explicará detalladamente el algoritmo de reconocimiento de caracteres empleado en este proyecto. Podemos dividir la implementación en dos partes diferenciadas: por un lado el procesamiento de la imagen necesario para poder separar los caracteres; y por otro lado el reconocimiento mediante una red neuronal.

3.1. Fase 1: Segmentación de caracteres

Como hemos introducido antes, dividimos el algoritmo de reconocimiento de caracteres en dos fases. En esta primera fase buscamos separar cada uno de los distintos caracteres de nuestra imagen. El flujograma de la figura 3.1 indica el proceso.

Nuestro punto de partida será una imagen digital del texto a reconocer. El primer paso es cargar la imagen en niveles de gris mediante la función `cvLoadImage` (en el Apéndice C se explican detalladamente cada una las funciones empleadas).

A continuación aplicamos un filtro a la imagen (`cvSmooth`) para eliminar posible ruido presente. Las imágenes con las que hemos trabajado son de una calidad elevada, por lo que no se aprecia un gran cambio entre la imagen inicial y la final. Suavizamos con un filtro gaussiano de 3x3, aunque también obtenemos resultados correctos con otras opciones como filtro de la mediana, bilateral... Máscaras más pequeñas no realizan ningún efecto y más grandes perjudicamos los contornos de los caracteres.

Una vez eliminado el ruido, procedemos a binarizar la imagen mediante la función `cvThreshold`. El valor umbral se calcula mediante el método Otsu para cada imagen (teniendo en cuenta la varianza de los niveles de gris de la misma), obteniendo resultados mucho mejores que con un valor umbral fijo.

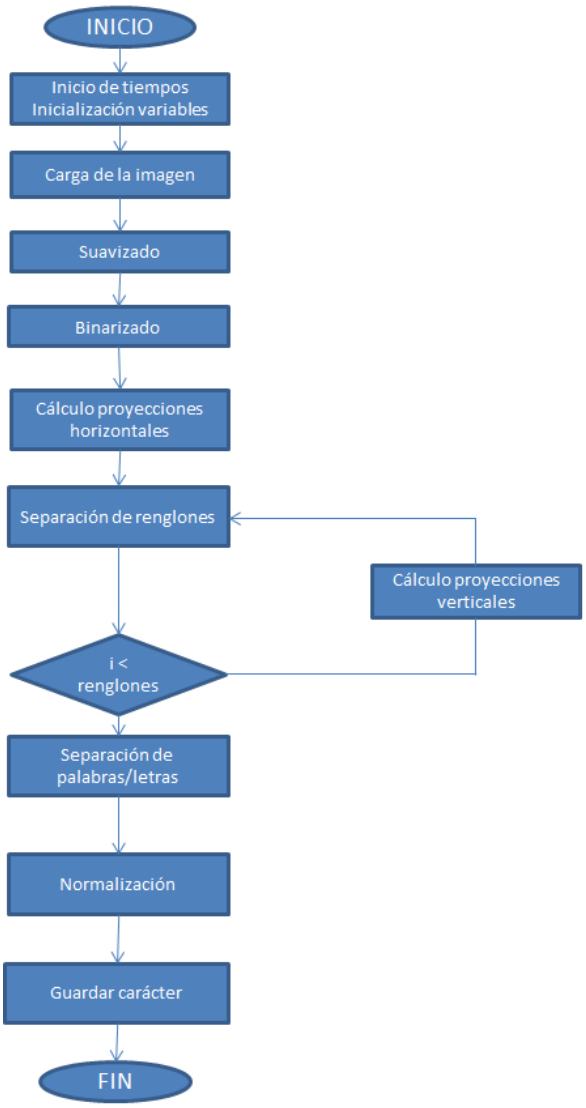

Figura 3.1: Flujograma fase 1

En este punto se trató de adelgazar los caracteres mediante erosiones y dilataciones, pero no obtuvimos resultados satisfactorios. En todos los casos perdíamos información relevante y los caracteres se volvían irreconocibles.

El objetivo final de esta fase es conseguir "recortar" cada uno de los caracteres presentes en la imagen. Para ello, el primer paso es detectar los renglones del texto. Calculamos las proyecciones horizontales de la imagen (función `proyeccionh`) y consideramos un renglón cuando encontramos 5 o más "filas" consecutivas de la imagen con una proyección menor de un valor fijado. Este valor fue fijado empíricamente, aunque no requiere una estimación demasiado precisa, ya que la diferencia en la proyección

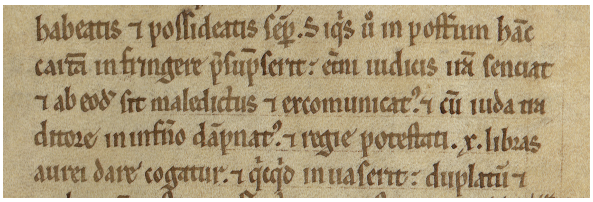

Figura 3.2: Imagen original

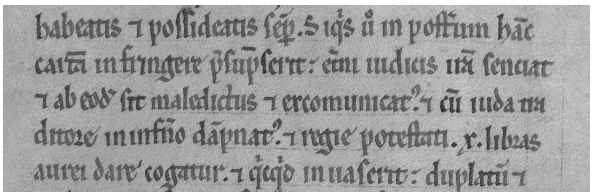

Figura 3.3: Imagen en niveles de gris

horizontal entre un renglón sin caracteres y otro con caracteres es más que notable.
Se estudió la posibilidad de incluir una detección de bordes de Canny antes del cálculo de proyecciones (buscando cierta inmunidad respecto al grosor del trazo), pero no obtuvimos buenos resultados, debido a que los caracteres quedan reducidos a muy pocos píxeles y su influencia en el valor medio de las proyecciones era imperceptible. Además encontramos problemas de falsos bordes. Volviendo a las proyecciones, de esta forma tenemos las coordenadas x de inicio y fin de cada uno de los renglones, que ampliamos en 5 píxeles para dar cabida a los trazos de letras como "p", "d"...
Realizamos un proceso análogo con las proyecciones verticales (función `proyeccionv`), pero dentro de cada renglón para separar las letras. La separación entre dos letras se dará cuando aparece una proyección vertical por encima de un valor fijo calculado también empíricamente. En este caso este valor fijo requiere más precaución. Al tratarse de un trazo realizado con pluma en algunos casos, no es uniforme y podemos encontrar trazos de unión entre distintas letras más gruesos que el propio trazo de una de ellas, por lo que partiríamos un carácter incorrectamente. Este hecho se ha intentado minimizar mediante ensayo y error, pero no se ha podido evitar. También calculamos un valor medio de anchura de caracteres, de tal forma que si hemos detectado un caracter anormalmente ancho, lo dividimos en partes de anchura media (porque al tratarse de texto manuscrito, en ocasiones varias letras pueden detectarse como una sola).
Mediante este proceso de proyecciones hemos almacenado las 4 coordenadas del rectángulo que delimita cada uno de los caracteres. Como los renglones se ampliaron en varios píxeles, debemos eliminar las zonas "vacías" de nuestro rectángulo, ya que en caso contrario la normalización de la imagen resultará en un caracter ilegible. Para esto, volvemos a utilizar las proyecciones horizontales dentro del rectángulo. De esta manera

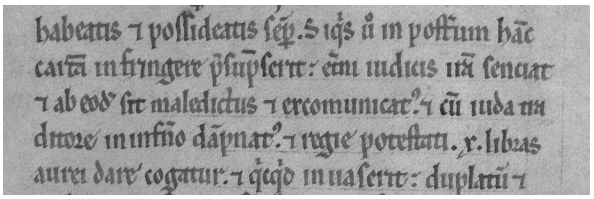

Figura 3.4: Imagen suavizada (filtro gaussiano)

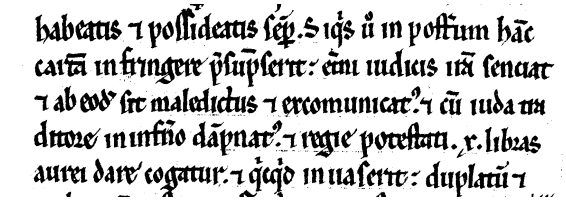

Figura 3.5: Imagen binarizada

conseguiremos enmarcar con más precisión nuestro caracter.

Figura 3.6: Resultado segmentación caracteres

Llegados a este punto, podremos introducir los caracteres ya extraídos de la imagen en la red neuronal. Pero en primer lugar debemos crear y entrenar dicha red. Obviamente este paso únicamente se realiza una vez, pero para ello se extraen los caracteres de varias imágenes distintas para poder tener una muestra amplia con la que formaremos el conjunto de aprendizaje.

3.2. Fase 2: Red neuronal

Una vez que hemos conseguido segmentar los caracteres de la imagen, podemos pasar al reconocimiento mediante una red neuronal. El flujograma de esta fase es el que podemos ver en la figura 3.7.

En primer lugar realizamos el entrenamiento de la red. Para el entrenamiento de la red, necesitamos que cada una de las imágenes esté nombrada de una manera parti-

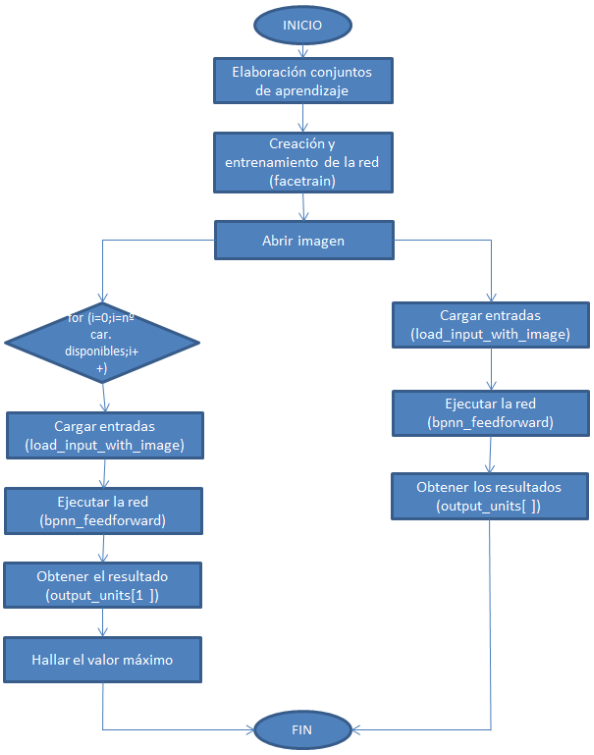

Figura 3.7: Flujograma fase 2

cular, ya que el nombre de archivo es lo que le permite a la red saber de qué caracter se trata. La manera de nombrar las imágenes ha sido la siguiente: *caracter_tamaño_-cardinal.pgm*. Por ejemplo: *d_min_3.pgm*. Estos caracteres de obtuvieron de distintos textos utilizando el código de la fase anterior. La elección de estos textos fue aleatoria, aunque se intentó buscar aquéllos con mejores tipografías. Se hicieron varias pruebas de conjuntos de entrenamiento, de las que hablaremos más adelante. Los textos elegidos aparecen en las Figuras 3.8, 3.9 y 3.10.

Se dividen las imágenes en 3 conjuntos: entrenamiento $(\frac{4}{9})$, validación $(\frac{2}{9})$ y test $(\frac{1}{3})$. Los dos primeros se usan en el aprendizaje de la red y el tercero nos sirve para evaluar el resultado obtenido.

Para ello, se elige un conjunto representativo de caracteres extraídos de distintas imágenes y se crea y entrena una red neuronal de 3 capas mediante el algoritmo de retropropagación basándonos en un código de la Carnegie Mellon University (`facetrain`, del que ya hemos hablado).
Se han analizado dos opciones para la red neuronal: 1 y 15 salidas (el número de caracteres distintos con los que hemos trabajado). Su implementación cambia ligeramente,

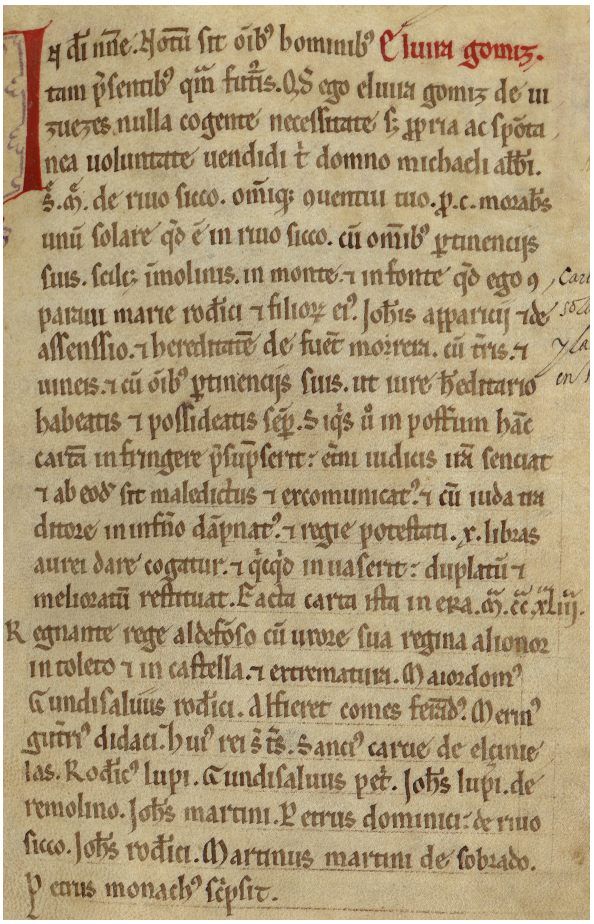

Figura 3.8: Imagen para aprendizaje: A17r.png

pero el fundamento es el mismo: dada la imagen de un caracter, lo introducimos en la red y evaluamos los resultados. Con redes de una salida, creamos 15 redes, una por caracter reconocible. En un bucle introducimos la imagen en todas ellas y tomamos la red que produzca la mayor salida como cierta. Con una sola red hacemos lo mismo, pero esas salidas se obtienen introduciendo la imagen una sola vez en la red.
Por tanto, podemos resumir el algoritmo completo en los siguientes pasos:

- Tratamiento de la imagen:
  - Suavizado
  - Binarizado
- Extracción de caracteres mediante proyecciones:
  - Renglones
  - Letras
- Reconocimiento mediante red neuronal:

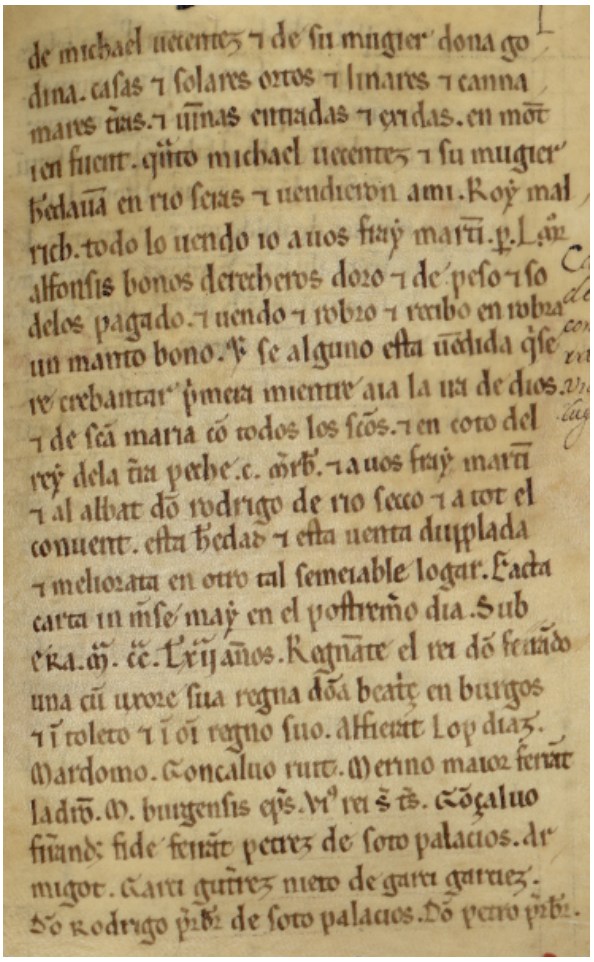

Figura 3.9: Imagen para aprendizaje: A56r.png

- Variante red neuronal 1 salida
- Variante red neuronal 15 salidas

En paralelo encontraríamos el proceso de creación y entrenamiento de la red neuronal, ya comentado.

Sobre este algoritmo se realizaron varias pruebas para determinar la influencia de algunos factores sobre los resultados obtenidos, que analizaremos en el capítulo de Experimentación y resultados.
Lógicamente, hay formas de mejorar este algoritmo. Algunas posibles alternativas se analizarán en el capítulo de Conclusiones y trabajos futuros.

Trabajando con `facetrain` Ya se ha comentado que la red neuronal empleada proviene de un código de la Universidad Carnegie Mellon. En este apartado se explicará en mayor detalle cómo está estructurado dicho código. Se trata de una red neuronal de 3

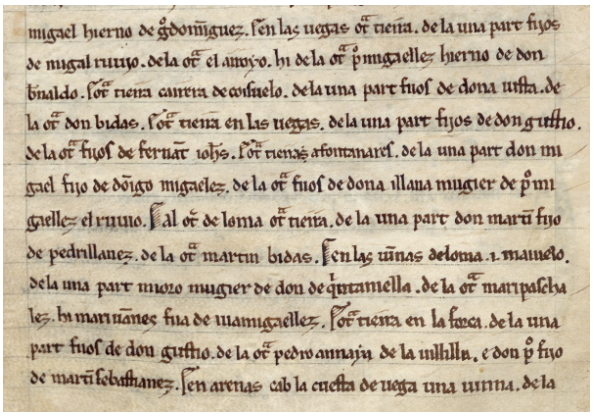

Figura 3.10: Imagen para aprendizaje: texto6.png

capas completamente conectadas con conexiones hacia delante (feedforward), que usa el algoritmo de retropropagación para ajustar sus pesos. También incluye un paquete de imágenes, un programa de alto nivel para entrenar y comprobar la red y una utilidad para visualizar el peso de los nodos ocultos. Después de compilar, obtenemos un ejecutable llamado `facetrain`. Este programa toma ficheros de listado de imágenes como entrada y los utiliza para entrenar y testear la red neuronal. También se utilizará en el reconocimiento y puede guardar las redes como ficheros *.net.
Módulos del código:

- `pgmimage.c`, `pgmimage.h`: este módulo de imágenes soporta la lectura/escritura de ficheros de imagen PGM y acceso/asignación a píxeles. Genera estructuras de datos IMAGE e IMAGELIST.
- `backprop.c`, `backprop.h`: el módulo de la red neuronal. Como hemos dicho antes, trabaja con redes de 3 capas completamente interconectadas, ajustando los pesos de los nodos mediante el algoritmo de retropropagación.
- `imagenet.c`: rutinas para cargar imágenes como entradas a una red neuronal y establecer los vectores objetivo para el entrenamiento.
- `facetrain.c`: es el programa de alto nivel que utiliza todos los módulos anteriores para implementar el reconocimiento.
- `hidtopgm.c`: la utilidad de visualización de los nodos ocultos.

Ejecutando `facetrain`

`facetrain` tiene muchas opciones que se especifican desde la línea de comandos a la hora de ejecutarlo. Aquí describiremos brevemente cómo funcionan:

- `-n <network file>`: esta opción carga una red neuronal o crea una nueva si no existe. Cuando finaliza el entrenamiento, se guarda la red neuronal en este fichero.
- `-e <number of epochs>`: Representa el número de iteraciones (epochs) que se realizarán. Por defecto es 100.
- `-T`: modo de test únicamente (sin entrenamiento). Se presenta el rendimiento de cada uno de los tres conjuntos de datos especificados, y las imágenes mal clasificadas se listarán junto con los niveles de sus salidas.
- `-s <seed>`: un entero que se utiliza como "semilla" del generador de números aleatorios. Por defecto es 102194. Esto permite reproducir experimentos generando la misma secuencia de números aleatorios, o probar con otra secuencia distinta.
- `-S <number of epochs between saves>`: esta opción especifica el número de iteraciones que se realizan hasta guardar la red neuronal. Por defecto es 100, lo que significa que la red se guardará cada 100 iteraciones.
- `-t <training image list>`: especifica el fichero de texto en el que se encuentra la lista de rutas de imágenes que componen el conjunto de entrenamiento.
- `-1 <test set 1 list>`: análogamente a la opción anterior, indica el fichero de texto que contiene el conjunto de validación.
- `-2 <test set 2 list>`: indica el fichero de texto que contiene el conjunto de test, con el que se realiza una prueba real de la red neuronal; no se modifican pesos en la red.

Interpretando la salida de `facetrain`

En primer lugar, `facetrain` imprime por pantalla unas líneas sobre los datos que está leyendo. Una vez que ha cargado los datos, empieza con el entrenamiento. En cada iteración, `facetrain` muestra por pantalla el rendimiento del entrenamiento y test de la red neuronal, mediante las siguientes medidas:
`<epoch><delta><trainperf><trainerr><t1perf><t1err><t2perf><t2err>`

- `epoch`: el número de la iteración que se acaba de completar.
- `delta`: es el sumatorio de todos los valores δ en las neuronas ocultas y de salida según se calcula en el algoritmo de retropropagación, sobre todos los ejemplos de entrenamiento para esa iteración.

- `trainperf`: es el porcentaje de ejemplos en el conjunto de entrenamiento que fueron correctamente clasificados.
- `trainerr`: es la media, sobre todos los ejemplos de entrenamiento, de la función error $\frac{1}{2}\sum(t_i - o_i)^2$, donde t_i es el valor objetivo para la neurona de salida i y o_i es el valor real para esa neurona.
- `t1perf`: es el porcentaje de ejemplos del conjunto de validación correctamente clasificados.
- `t1err`: análogo a `trainerr`, pero para el conjunto de validación.
- `t2perf`: porcentaje de ejemplos del conjunto de test que fueron completamente clasificados.
- `t2err`: al igual que `trainerr` y `t1err`, representa la media del error cometido en el conjunto de test.

Módulo de la red neuronal

Como ya se ha dicho anteriormente, este módulo implementa una red neuronal de 3 capas completamente interconectadas realimentada hacia delante, utilizando el método de retropropagación para ajustar los pesos. En primer lugar, describimos brevemente la estructura de datos `BPNN` (Back Propagation Neural Network).

Todos los valores de las neuronas y sus pesos se almacenan como `double` en `BPNN`. Dada una estructura `BPNN *net`, podemos obtener el número de neuronas de entrada, ocultas y de salida mediante `net->input_n`, `net->hidden_n` y `net->output_n`. Las neuronas se numeran de 1 a n, siendo n el número de neuronas en una capa. Para obtener el valor de la k-ésima neurona en cualquiera de las capas (entrada, ocultas o de salida), utilizamos `net->input_units[k]`, `net->hidden_units[k]` y `net->output_units[k]`.

El vector objetivo debe tener tantos valores como neuronas tenga la capa de salida, y se puede acceder a el mediante `net->target`. El valor k-ésimo del vector objetivo lo obtenemos mediante `net->target[k]`.

Para obtener el valor del peso de la conexión entre la neurona de entrada i y la neurona oculta j, utilizamos `net->input_weights[i][j]`. Si es el peso entre una neurona oculta j y la neurona de la capa de salida k, `net->hiddenweights[j][k]`.

Capítulo 4

Experimentación y resultados

4.1. Red Neuronal

Se han realizado 4 pruebas referentes a los parámetros de la red neuronal. Con estas pruebas buscábamos analizar 2 efectos principales: capacidad de reconocimiento de la red y rapidez del sistema. Para ello, modificamos los siguientes parámetros de la red:

- Número de salidas
- Número de entradas
- Modificación conjuntos aprendizaje
- Variación Tasa de aprendizaje y Momento de la red

A continuación se explicarán detalladamente cada una de las pruebas y se analizarán los resultados obtenidos.

4.1.1. Prueba 1: Modificación número de salidas de la red neuronal

En esta primera prueba analizamos las dos posibles soluciones que encontramos al problema del reconocimiento de caracteres: una sola red con tantas salidas como caracteres pudiéramos reconocer o tantas redes como caracteres, cada una de ellas capaz de reconocer un solo caracter. Se expondrán los resultados para los caracteres "a" y "b". Las redes neuronales creadas tienen 100 entradas (las imágenes empleadas en el aprendizaje están normalizadas a unas dimensiones de 10x10 píxeles).
Para esta prueba se ha empleado un conjunto de 205 imágenes para el aprendizaje, divididas en 3 grupos: entrenamiento (91), validación (45) y test (69). Este conjunto

	Redes 1 salida		Red 15 salidas	
a	0.8866	OK	0.8689	OK
b	0.2157	FAIL	0.8875	FAIL
d	0.8459	OK	0.9452	OK
de	0.8976	OK	0.9020	FAIL
e	0.9178	OK	0.8874	OK
espacio	0.8716	OK	0.8773	OK
i	0.3543	FAIL	0.7219	FAIL
l	0.4208	FAIL	0.5889	OK
n	0.5576	FAIL	0.9626	FAIL
o	0.8695	OK	0.8864	OK
p	0.1277	FAIL	0.8447	FAIL
punto	0.3910	FAIL	0.7971	FAIL
r	0.6861	FAIL	0.3193	FAIL
s	0.8745	OK	0.9068	OK
u	0.3957	FAIL	0.8409	FAIL

Cuadro 4.1: Salidas obtenidas para las distintas redes y evaluación de resultados

se utilizará en casi todas las pruebas siguientes. Por simplicidad, se han utilizado en todos los casos 500 iteraciones en el aprendizaje y se han anotado varios tiempos de ejecución. De esta forma, podemos analizar posteriormente el error cometido durante el entrenamiento por la red y determinar cuánto tiempo necesita dicha red para convergir.

A continuación analizaremos la evolución del error cometido por la red durante el entrenamiento. Para tener una idea más representativa, también mostraremos el error medio cometido por la red con los grupos de aprendizaje que utilizamos para evaluar nuestros resultados: validación y test. En todos los casos hemos empleado 500 iteraciones y se han registrado varias medidas del tiempo transcurrido para poder estimar la rapidez con la que converge la red.
En la figura 4.1 mostramos las gráficas para redes de 1 y 15 salidas. Para las redes de 1 salida se han elegido dos caracteres al azar, en este caso *a* y *b*.

Para estimar la velocidad de convergencia de las redes de una sola salida, tomamos varias medidas de tiempo para 500 iteraciones, y observando las tendencias en los errores podemos interpolar para conocer el momento en el que consideramos que la red converge. La red para el caracter "a" obtuvo una media de 1123.957 ms y podemos

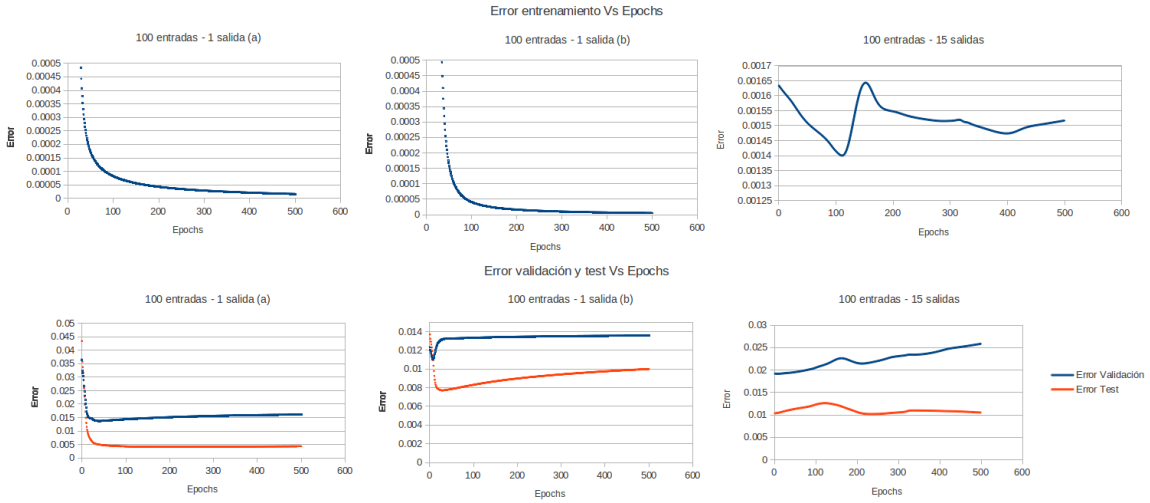

Figura 4.1: Errores para redes de 1 y 15 salidas

suponer que converge a partir de la iteración número 500. En el caso de la red "b", converge en unas 400 iteraciones, por lo que con una media de 1140.141 ms estimamos su convergencia a los 912.113 ms. La red "a" es algo más lenta, pero obtiene mejores errores medios (0.005 frente a 0.01 en el conjunto de test).

Para el caso de la red de 15 salidas se realizaron 4 mediciones de tiempo: 1680.178, 1548.709, 1535.411 y 1585.778 ms. Tomamos la media: 1587.519 ms. El grupo de validación nos da una idea de cómo de bueno está siendo nuestro entrenamiento, motivo por el que los errores de entrenamiento y validación están relacionados. En este caso, ninguno de los dos converge en las 500 iteraciones, por lo que sería más apropiado usar una condición de parada de error mínimo. En una de las múltiples pruebas paralelas que se realizaron, con conjuntos de aprendizaje ligeramente distintos, obtuvimos la gráfica representada en la figura 4.2.

Uno de los principales problemas del aprendizaje de una red neuronal es el sobreentrenamiento, ya que perdemos toda capacidad de generalización. La red se especializa demasiado en su propio conjunto de aprendizaje y no obtiene buenos resultados cuando se enfrenta a casos distintos. En la figura 4.2 aparece una oscilación del error de test a partir de un número de iteraciones. El sobreentrenamiento se traduce en esta inestabilidad y en un empeoramiento de los resultados.

Las redes de una sola salida son mucho más rápidas, lo cual era de esperar por el menor número de pesos con los que tienen que trabajar. En cuanto al error, en la red de 15 salidas obtenemos un error medio en el conjunto de test similar al del caracter

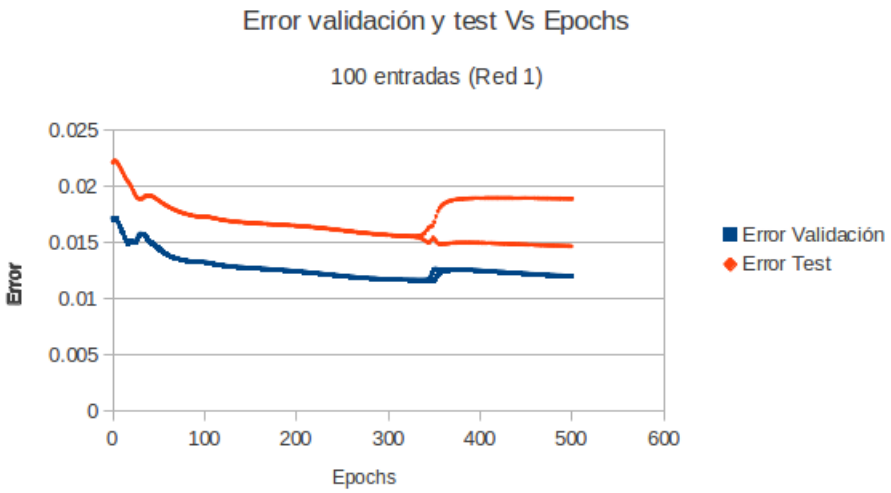

Figura 4.2: Fenómeno de sobreentrenamiento

"b" (una red con resultados pobres), pero dos veces el error medio del caracter "a".

4.1.2. Prueba 2: Variación del número de entradas de la red

En la primera prueba analizábamos dos enfoques del problema que repercutían en el algoritmo empleado para el reconocimiento: redes con una salida, o una sola red de 15 salidas. Ahora estudiaremos la influencia del número de entradas utilizado en el rendimiento de la red. Es un factor importante, ya que a mayor número de entradas, mayor información disponible para el aprendizaje, pero mayor complicación del mismo y tiempo de proceso. Hemos creado redes de 16, 25, 49 y 100 entradas (normalizando las imágenes de los conjuntos de aprendizaje a 4x4, 5x5, 7x7 y 10x10 píxeles respectivamente). Al igual que en la Prueba 1, se han realizado 500 iteraciones en el entrenamiento de la red.

Después de analizar los resultados mostrados en las figuras 4.3, 4.4, 4.5 y 4.6, se observa una tendencia distinta entre distintos caracteres. En nuestro caso, el caracter "a" mejora en ambas redes (1 ó 15 salidas) según aumentamos el número de entradas, después de haber encontrado un pequeño mínimo. En cambio, el caracter "b" presenta la tendencia contraria. Esto puede ser debido a las limitaciones en el algoritmo de segmentación de caracteres empleado, ya que aparecen muchos píxeles erróneos de caracteres adyacentes o por simples imperfectos en el trazo. Depurando dicha segmentación y consiguiendo un conjunto de aprendizaje libre de "desperfectos" es probable que consiguiéramos reducir este efecto.

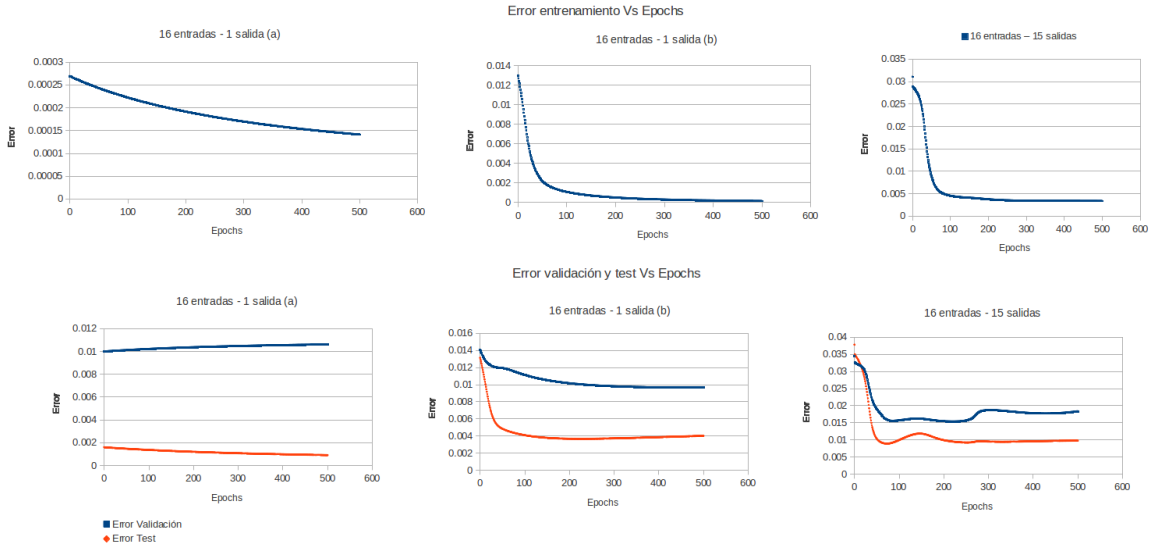

Figura 4.3: Errores para red de 16 entradas

Además, aumentar el número de entradas puede suponer demasiada información para la red neuronal. Con los 15 caracteres estudiados es posible que 100 entradas sean excesivas y consigamos entorpecer a la red en lugar de mejorarla.

Otro aspecto encontrado (común a todas las pruebas) es que hay caracteres similares que pueden llegar a "confundir" a la red. Es decir, tenemos una red que reconoce una "b" como una "s". Si eliminamos la "s" del conjunto de aprendizaje, reconoce la "b" como "b". En este caso deberíamos buscar alguna extracción de características adicional, o emplear la idea que se explica en el primer capítulo sobre el reCAPTCHA, en el que es una persona quien reconoce el caracter correcto en los casos en los que el método no consigue buenos resultados.

Como última observación general a las pruebas, el orden de las imágenes en los conjuntos de aprendizaje influye en los resultados finales, pero no se ha estudiado en detalle por la gran complejidad que supone encontrar la combinación más óptima. Simplemente sabemos que los resultados pueden variar, pero lo consideramos una componente aleatoria.

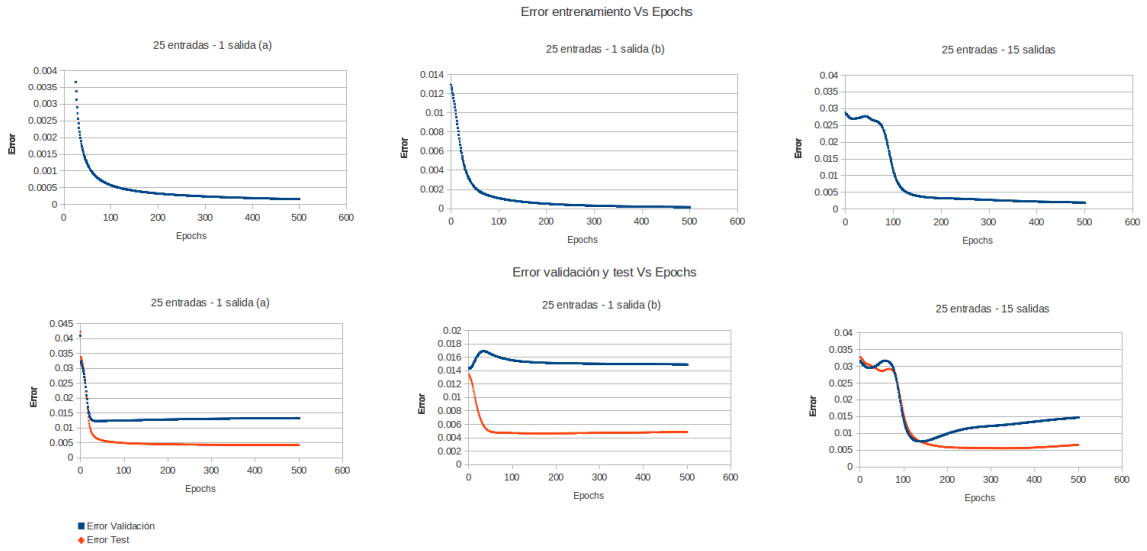

Figura 4.4: Errores para red de 25 entradas

	Conjuntos de aprendizaje				
	1	2	3	4	5
Entrenamiento	90	106	92	31	51
Validación	45	54	46	19	25
Test	68	78	70	22	39

Cuadro 4.2: Distribución de los conjuntos de aprendizaje

4.1.3. Prueba 3: Modificación de conjuntos de aprendizaje

Nuestro objetivo es determinar si la elección de unas u otras imágenes para los conjuntos de aprendizaje de nuestra red, así como el tamaño de la muestra tomada, influyen de manera sustancial en los resultados obtenidos. En esta prueba, se han elaborado 5 conjuntos de aprendizaje: 3 de ellos de aproximadamente el mismo número de muestras, pero con una población distinta; los otros dos son de tamaño menor. En el Cuadro 4.2 podemos ver la distribución de grupos.

En las figuras 4.7 y 4.8 tenemos las gráficas de error de los distintos conjuntos: se han analizado los 5 conjuntos, el 1 y el 2 únicamente en la red de 15 salidas y los conjuntos 3, 4 y 5 en las redes de 1 salida (caracteres "a" y "b") y 15 salidas.
Al analizar los resultados, se observa un aumento del error cometido según disminuimos los conjuntos de aprendizaje. El error cometido en los conjuntos 4 y 5 es mayor que en los grupos 1, 2 y 3. Por lo demás, podemos afirmar que la muestra escogida sí afecta a

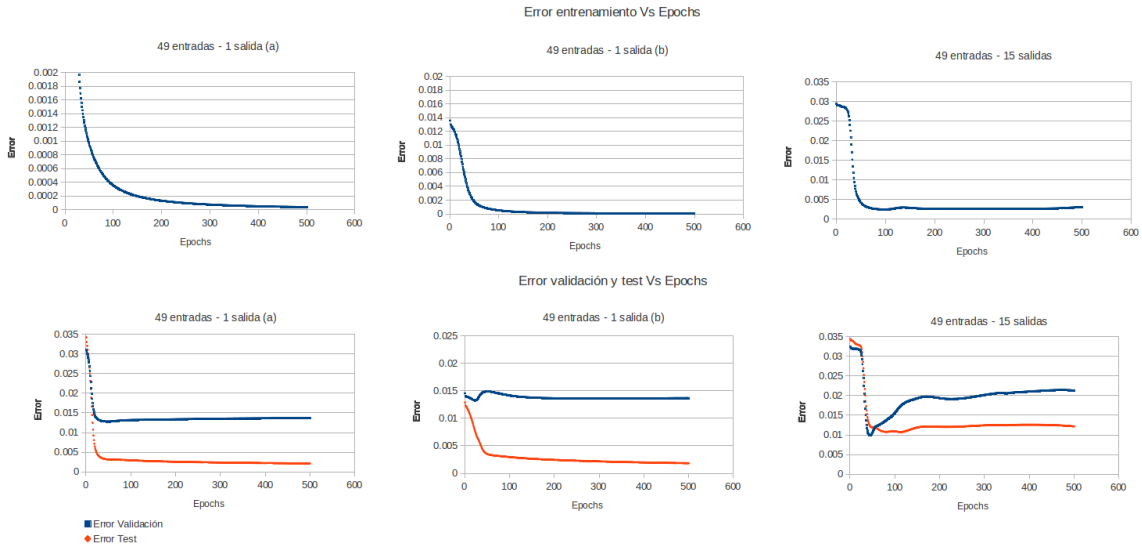

Figura 4.5: Errores para red de 49 entradas

la capacidad de reconocimiento de nuestra red. Los conjuntos 1, 2 y 3 están formados por aproximadamente el mismo número de elementos, pero los errores encontrados son muy distintos. La tendencia del error correspondiente a cada conjunto es diferente: en el conjunto 1 se reduce gradualmente hasta llegar a un punto de inestabilidad, por sobreentrenamiento de la red; el conjunto 2 es mucho más inestable, llegando a un mínimo a partir del cual los errores de validación y test crecen monótonamente; y en el conjunto 3 observamos un comportamiento relativamente plano, pero ligeramente ascendente en el caso del error de validación. Por tanto podemos afirmar que la elección de las imágenes que forman cada conjunto influyen de manera importante en los resultados de la red.
Para el conjunto 3 se han analizado las dos tipologías de red neuronal: 1 y 15 salidas. Observamos un comportamiento más plano en las redes de 1 salida, además de un error menor.

En la figura 4.9 se muestra la salida obtenida en el algoritmo de reconocimiento para los caracteres ”a” y ”b”, frente al número de entradas.

Observamos un efecto completamente distinto entre los dos caracteres. Para el caracter ”a”, el reconocimiento mejora según aumenta el número de entradas. En cambio, para el caracter ”b”, empeora. La explicación encontrada a este hecho es la ”mala” extracción de dichos caracteres de la imagen. Al tratarse de un texto manuscrito, la segmentación de los caracteres puede llegar a tener resultados aleatorios: hay carac-

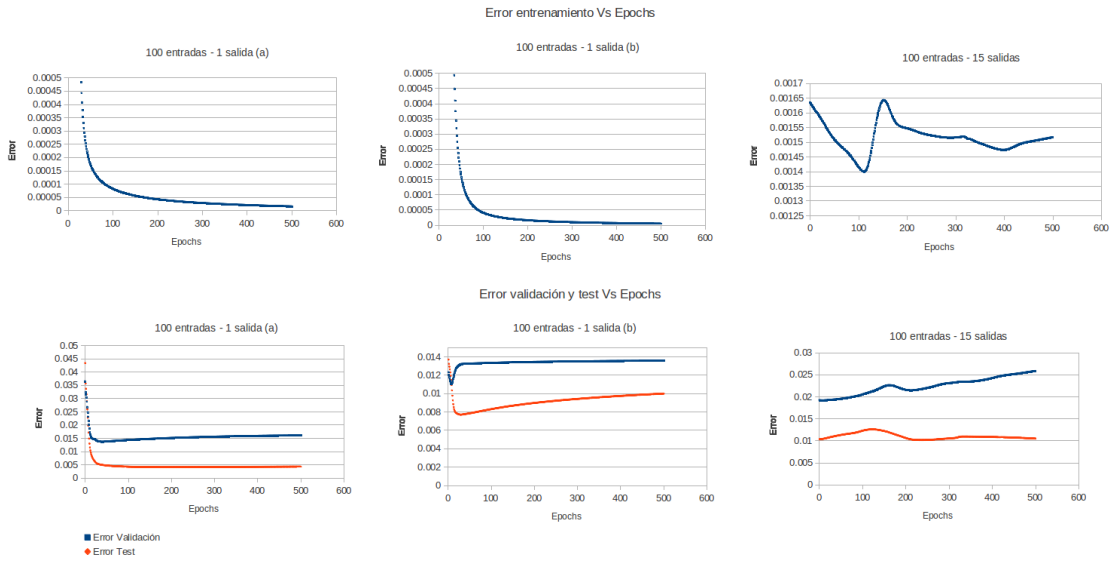

Figura 4.6: Errores para red de 100 entradas

teres muy bien extraídos y otros no tanto. Este hecho afecta a los resultados, ya que un caracter con una mala extracción de la imagen difícilmente puede conseguir buenos resultados si la red está entrenada con "buenos" caracteres, y viceversa.

4.1.4. Prueba 4: Modificación tasa de aprendizaje y momento

En esta prueba, buscamos analizar el efecto de estos parámetros en el comportamiento de la red. La tasa de aprendizaje nos da una idea de cuánto se desplazan los pesos de la red sobre la superficie del error en la dirección negativa del gradiente. Valores bajos de esta tasa de aprendizaje pueden hacer que la convergencia sea excesivamente lenta, mientras que valores altos pueden hacer que nos saltemos el mínimo del error o que oscilemos alrededor de él. El momento controla la "inercia" del sistema: añade una fracción de la variación del peso en la iteración anterior a la actual. De manera parecida a la tasa de aprendizaje, valores altos del momento pueden hacer que el sistema sea inestable y valores bajos pueden hacer que caigamos en mínimos locales. Se han realizado pruebas manteniendo uno de los dos parámetros (0.3) y variando el otro (0.05, 0.3, 0.7 y 0.95). Todas las pruebas se han realizado con el mismo conjunto de aprendizaje, ya que únicamente nos interesa evaluar el efecto de estos dos parámetros. Como en todas las pruebas, se realizan 500 iteraciones.

En primer lugar analizaremos la variación de la tasa de aprendizaje. Observando la subfigura a) dentro de la figura 4.10 comprobamos que con una tasa de 0.05, la disminución del error con las iteraciones es muy lenta, no llegando a converger en el

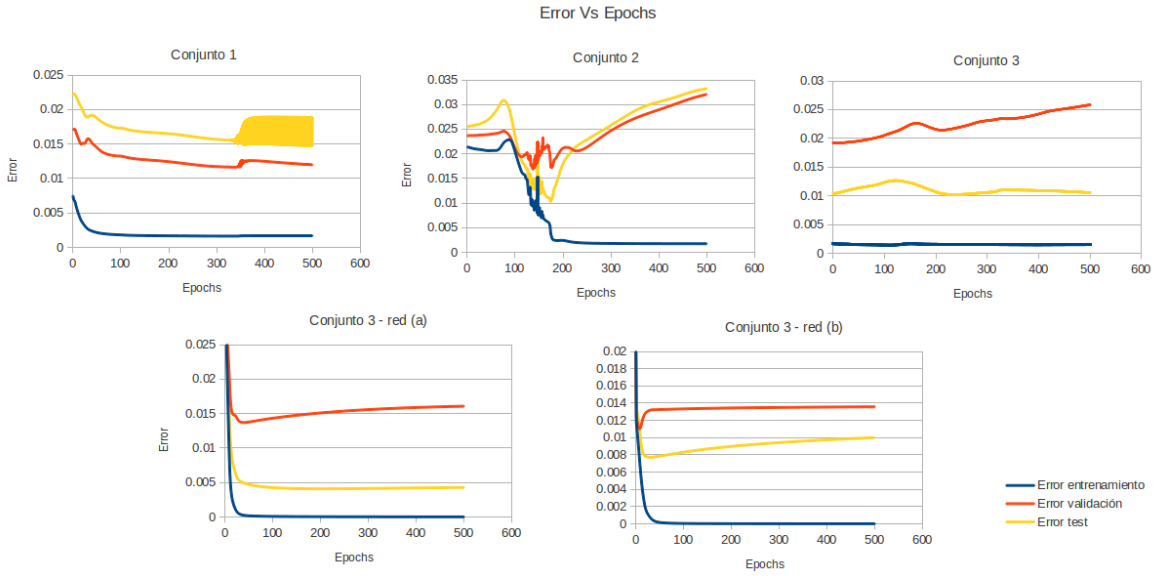

Figura 4.7: Error con distintos conjuntos de aprendizaje (i)

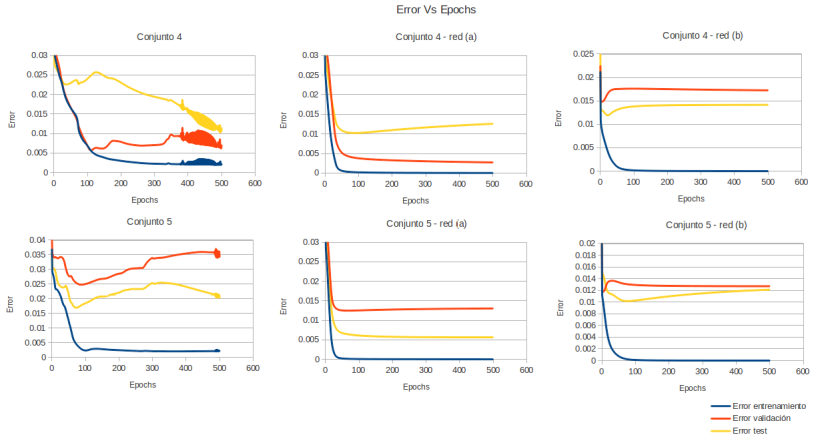

Figura 4.8: Error con distintos conjuntos de aprendizaje (ii)

transcurso de las 500 iteraciones. De esta forma sabemos que no nos saltaremos el punto de error mínimo, pero perjudicamos considerablemente la rapidez del sistema.

En la figura 4.10, subfigura b), vemos que con una tasa de aprendizaje de 0.3 conseguimos una convergencia mucho más rápida, en unas 200 iteraciones (633.585 ms).

Podemos ver en la subfigura c) que una tasa de aprendizaje de 0.7 inestabiliza nuestro sistema. No podemos considerar una convergencia real, aunque los valores de error estén acotados en un entorno de unas 0.0005 unidades.

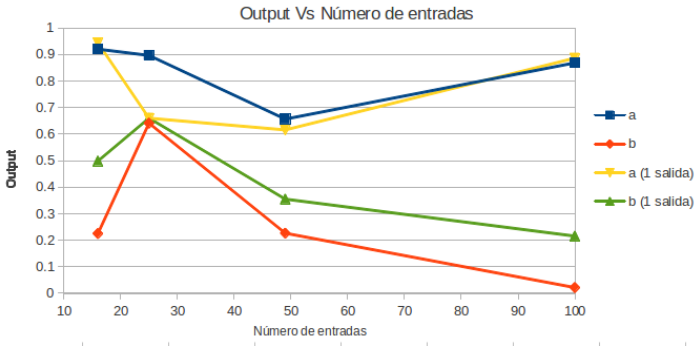

Figura 4.9: Salida obtenida del algoritmo frente a número de entradas de la red

El comportamiento de la red con una tasa de 0.95 (figura 4.10, subfigura d)) es muy similar al de tasa igual a 0.7. Existe un valor límite de la tasa de aprendizaje a partir del cual el sistema es inestable y por ello los resultados para estas dos tasas son casi idénticos. Podemos estimar que ese valor límite esté comprendido entre 0.5-0.6. De esta forma se confirma lo visto en la parte teórica sobre redes neuronales. Hasta ese valor, conforme aumentamos la tasa de aprendizaje obtenemos convergencias más rápidas.

A continuación analizamos el impacto de la variación del momento en los resultados. Con un momento de 0.05 (figura 4.10, subfigura e)) la inercia del sistema es tan baja a la hora de actualizar los valores de los pesos que la convergencia es muy lenta, tanto que no podemos encontrarla en 500 iteraciones.
Para un momento de 0.3 (subfigura b)) obtenemos la gráfica ya comentada en el párrafo anterior, consiguiendo convergir en 200 iteraciones aproximadamente. Podemos considerar esta combinación como la más equilibrada a la hora de elegir entre velocidad de convergencia y error.
De manera análoga a lo visto con la tasa de aprendizaje, valores altos del momento inestabilizan de gran manera el sistema, consiguiendo resultados poco satisfactorios (subfiguras e) y f)).

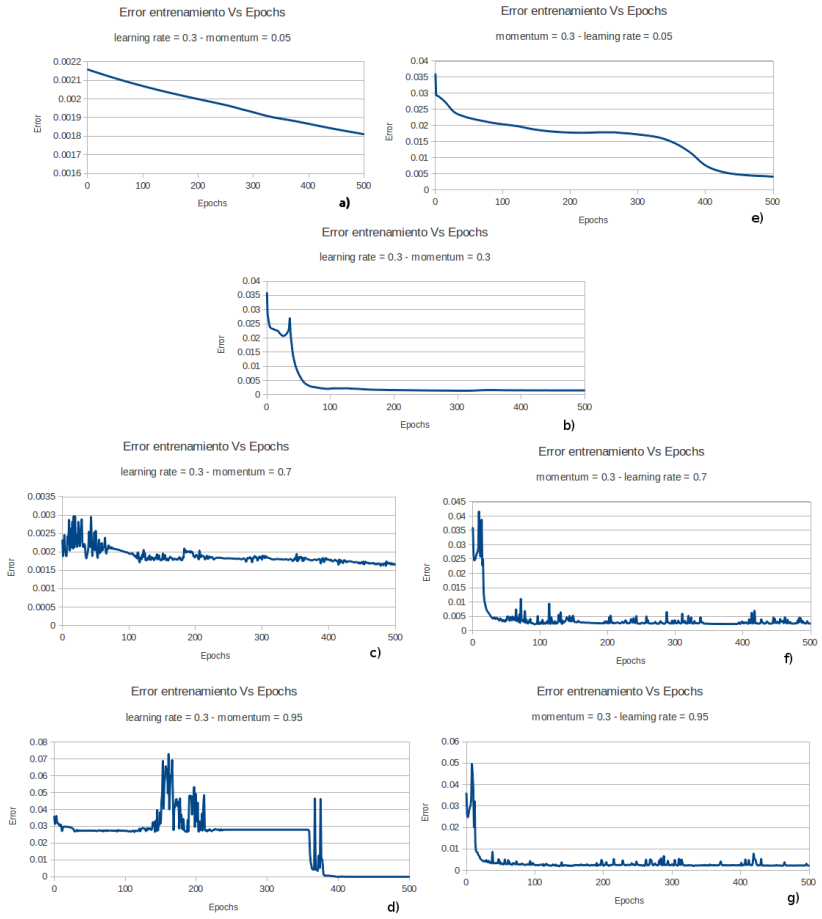

Figura 4.10: Variación tasa de aprendizaje y momento

Capítulo 5

Conclusiones y trabajos futuros

Conclusiones

Después de explicar todas las pruebas realizadas y exponer y analizar los resultados obtenidos, es el momento de sacar conclusiones sobre el trabajo. La principal conclusión es la importancia de una buena extracción de los caracteres de la imagen. Hemos encontrado muchos errores en la fase de reconocimiento mediante redes neuronales que son consecuencia de defectos en la segmentación de la imagen. Al tratarse de textos manuscritos, es un factor muy relevante, ya que es muy complicado elaborar algoritmos eficaces a la hora de separar correctamente cada caracter.

- En este proyecto se ha conseguido desarrollar un sistema de reconocimiento de caracteres antiguos diferenciando diferentes escenarios en el sistema de aprendizaje.
- Aunque inicialmente se pensó que una red neuronal con 15 salidas sería más general y por lo tanto más eficiente que 15 redes independientes se ha demostrado que no es cierto debido a la naturaleza del problema.
- Los tiempos de convergencia de una red con una salida son mucho menores que los de la red con 15 salidas. Los tiempos de ejecución son similares.
- Los resultados mejoran sustancialmente al eliminar caracteres confundibles geométricamente. Es decir, la correlación de caracteres afecta negativamente a la tasa de reconocimiento.
- La red responde correctamente en términos de convergencia mediante la variación de la tasa de aprendizaje y momento tal y como se esperaba.
- Un aumento del conjunto de entrada al sistema de aprendizaje ayuda a aumentar la tasa de reconocimiento.

Trabajos futuros

En este proyecto se propone el aprendizaje mediante NN. Se deja para futuros trabajos el aprendizaje mediante otras técnicas como KNN, SVM, Bayesian Netwoks, etc.

La extracción de los caracteres es un proceso clave. Se propone investigar nuevas técnicas de segmentación de caracteres que mejoren los datos de entrada al sistema de aprendizaje.

Se puede extender el aprendizaje a subgrupos de letras/palabras, añadiendo sílabas/palabras que permitan mejorar la tasa de aprendizaje y no enfocarse únicamente a caracteres sueltos.

Además, podemos incluir la capacidad para reconocer textos multicolumna, gráficos insertados, etc.

Apéndice A

Instalación OpenCV

Linux

Prebuilt binaries for Linux are not included with the Linux version of OpenCV owing to the large variety of versions of GCC and GLIBC in different distributions (SuSE, Debian, Ubuntu, etc.). If your distribution doesn't offer OpenCV, you'll have to build it from sources as detailed in the .../$opencv/INSTALL$ file. To build the libraries and demos, you'll need GTK+ 2.x or higher, including headers. You'll also need *pkgconfig*, *libpng*, *zlib*, *libjpeg*, *libtiff* , and *libjasper* with development files. You'll need Python 2.3, 2.4, or 2.5 with headers installed (developer package). You will also need *libavcodec* and the other *libav∗* libraries (including headers) from *ffmpeg* `0.4.9-pre1` or later (*svncheckoutsvn* : *//svn.mplayerhq.hu/ffmpeg/trunkffmpeg*). Download ffmpeg from *http* : *//ffmpeg.mplayerhq.hu/download.html*.The ffmpeg program has a lesser general public license (LGPL). To use it with non-GPL software (such as OpenCV), build and use a shared ffmpg library:

```
$>./configure --enable-shared
$>make
$>sudo make install
```

You will end up with: */usr/local/lib/libavcodec.so.∗*, */usr/local/lib/libavformat.so.∗*, */usr/local/lib/libavutil.so.∗*, and include files under various */usr/local/include/libav∗*.

NOTE: It is important to know that, although the Windows distribution contains binary libraries for release builds, it does not contain the debug builds of these libraries. It is therefore likely that, before developing with OpenCV, you will want to open the solution file and build these libraries for yourself. You can check out ffmpeg by: `svn checkout svn://svn.mplayerhq.hu/ffmpeg/trunk ffmpeg` To build OpenCV using Red Hat Package Managers (RPMs), use `rpmbuild -ta OpenCV-x.y.z.tar.gz`

(for RPM 4.x or later), or `rpm -ta OpenCV-x.y.z.tar.gz` (for earlier versions of RPM), where $OpenCV-x.y.z.tar.gz$ should be put in $/usr/src/redhat/SOURCES/$ or a similar directory. Then install OpenCV using `rpm -i OpenCV-x.y.z.*.rpm`.

To build OpenCV once it is downloaded:

```
$>./configure
$>make
$>sudo make install
$>sudo ldconfig
```

After installation is complete, the default installation path is $/usr/local/lib/$ and $/usr/local/include/opencv/$. Hence you need to add $/usr/local/lib/$ to $/etc/ld.so.conf$ (and run $ldconfig$ afterwards) or add it to the $LD\_LIBRARY\_PATH$ environment variable; then you are done. To add the commercial IPP performance optimizations to Linux, install IPP as described previously. Let's assume it was installed in $/opt/intel/ipp/5{,}1/ia32/$. Add $< yourinstall\_path > /bin/$ and $< yourinstall\_path > /bin/linux32$ `LD_LIBRARY_PATH` in your initialization script (*.bashrc* or similar):

```
LD_LIBRARY_PATH=/opt/intel/ipp/5.1/ia32/bin:/opt/intel/ipp/5.1 /ia32/
/bin/linux32:LD_LIBRARY_PATH
export LD_LIBRARY_PATH
```

Alternatively, you can add $< yourinstall\_path > /bin$ and $< yourinstall\_path > /bin/linux32$, one per line, to $/etc/ld.so.conf$ and then run $ldconfig$ as root (or use sudo). That's it.

Apéndice B

Funciones `facetrain`

A continuación veremos las funciones que aparecen en el código que crea nuestras redes neuronales, `facetrain`:

- `void bpnn_initialize(seed)`
  `int seed;`
  Esta función inicializa el módulo de la red neuronal, por lo que debe ser llamada en primer lugar. Realmente se limita a inicializar el generador de número aleatorio con la entrada `seed`.

- `BPNN *bpnn_create(n_in, n_hidden, n_out)`
  `int n_in, n_hidden, n_out;`
  Crea una red neuronal con `n_in` neuronas de entrada, `n_hidden` neuronas ocultas y `n_out` neuronas en la capa de salida. Los pesos en la red se asignan aleatoriamente con valores comprendidos en el intervalo [-1, 1]. Devuelve un puntero a la estructura `BPNN` o `NULL` si la función falla.

- `void bpnn_free(net)`
  BPNN *net
  Toma un puntero a una red y libera toda la memoria asociada a dicha red.

- `void bpnn_train(net, learning_rate, momentum, erro, errh)`
  `BPNN *net;`
  `double learning_rate, momentum;`
  `double *erro, *errh;`
  Dado un puntero a una red, ejecuta un ciclo del algoritmo de retropropagación. Asume que las neuronas de entrada y el vector objetivo han sido configurados correctamente. `learning_rate` y `momentum` son valores comprendidos entre 0 y

1. `erro` y `errh` son punteros a double, a las sumas de los valores de error δ en las neuronas de salida y ocultas, respectivamente.

- `void bpnn_feedforward(net)`
  `BPNN *net;`
  Dado un puntero a una red, la ejecuta con sus valores de entrada actuales.

- `BPNN *bpnn_read(filename)`
  `char *filename;`
  Dado un nombre de fichero, reserva espacio en memoria para una red, la inicializa con los pesos almacenados en el fichero de la red y devuelve un puntero a la nueva estructura `BPNN`. Devuelve `NULL` si hay un error.

- `void bpnn_save(net, filename)`
  `BPNN *net;`
  `char *filename;`
  Dado un puntero a una red y un nombre de fichero, guarda la red en dicho fichero.

- `backprop_face(trainlist, test1list, test2list, epochs, savedelta, netname, list_errors)`
  `IMAGELIST *trainlist, *test1list, *test2list;`
  `int epochs, savedelta, list_errors;`
  `char *netname;`
  Lee la red neuronal `netname` o la crea si no existe, y ejecuta el algoritmo de retropropagación con el conjunto de aprendizaje definido por las listas de imágenes dadas en `trainlist, test1list` y `test2list`. Realiza `epochs` iteraciones y guarda la red cada `savedelta` iteraciones.

- `performance_on_imagelist(net, il, list_errors)`
  `BPNN *net;`
  `IMAGELIST *il;`
  `int list_errors;`
  Calcula los errores cometidos por la red `net` en las imágenes presentes en la lista `il`.

- `load_input_with_image(img, net)`
  `IMAGE *img;`
  `BPNN *net;`
  Carga cada píxel de la imagen `img` como una entrada de la red neuronal `net`.

- `evaluate_performance(net, err)`
  `BPNN *net;`
  `double *err;`
  Esta función verifica si el reconocimiento de un patrón de aprendizaje es correcto o no. Al entrenar la red para que reconozca una característica determinada, compara el patrón y la salida producida por la red con dicha característica, existiendo cuatro casos:
  - Patrón a nivel alto y salida a nivel alto: el patrón presentaba la característica buscada y el reconocimiento ha sido correcto.
  - Patrón a nivel alto y salida a nivel bajo: el sistema no ha reconocido la imagen.
  - Patrón a nivel bajo y salida a nivel alto: el sistema da un "falso positivo".
  - Patrón a nivel bajo y salida a nivel bajo: el patrón no presentaba la característica buscada y la red ha producido correctamente una salida contraria.
- `load_target(img, net)`
  `IMAGE *img;`
  `BPNN *net;`
  Esta función determina si una imagen del entrenamiento debe producir un nivel alto o bajo, en función de si en su nombre de fichero encuentra un patrón concreto.
- `double drnd()`
  Genera un número aleatorio entre 0 y 1.
- `double dpnl()`
  Genera un número aleatorio entre -1 y 1.
- `squash(x)`
  `double x;`
  La función de activación (función sigmoide).
- `double *alloc_1d_dbl(n)`
  `int n;`
  `double **alloc_2d_dbl(m,n)`
  `int m, n;`
  Funciones para reservar espacio de memoria.
- `bpnn_randomize_weights(w, m, n)`
  `double **w;`

```
int m,n;
bpnn_zero_weights(w, m, n)
double **w;
int m,n;
```

Funciones que inicializan aleatoriamente o asigan valor cero a los pesos de las conexiones entre neuronas.

- `BPNN *bpnn_internal_create(n_in, n_hidden, n_out)`
  `int n_in, n_hidden, n_out;`
  Reserva la memoria necesaria para crear una red neuronal con `n_in` neuronas en la capa de entrada, `n_hidden` en la capa oculta y `n_out` en la capa de salida.

- `void bpnn_layerforward(l1, l2, conn, n1, n2)`
  `double *l1, *l2, **conn;`
  `int n1, n2;`
  Propagación de las entradas de cada capa: cada neurona recibe una entrada y produce una salida que depende del umbral (peso asociado) y de la función de activación escogida.

- `void bpnn_output_error(delta, target, output, nj, err)`
  `double *delta, *target, *output, *err;`
  `int nj;`
  `void bpnn_hidden_error(delta_h, nh, delta_o, no, who, hidden, err)`
  `double *delta_h, *delta_o, *hidden, **who, *err;`
  `int nh, no;`
  Cálculo de la función error

- `void bpnn_adjust_weights(delta, ndelta, ly, nly, w, oldw, eta, momentum)`
  `double *delta, *ly, **w, **oldw, eta, momentum;`
  Función que calcula el nuevo peso asociado a una conexión según la expresión vista en la ecuación 2.105.

Módulo de imagen

Este módulo proporciona una serie de funciones para trabajar con imágenes en formato PGM.

- `IMAGE *img_open(filename)`
  `char *filename;`
  Abre la imagen dado por `filename`, carga sus datos en una estructura `IMAGE` y

devuelve un puntero a dicha estructura. Como es habitual, devuelve `NULL` si hay un error.

- `IMAGE *img_creat(filename, nrows, ncols)`
  `char *filename;`
  `int nrows, ncols;`
  Crea una imagen en memoria, con el nombre de archivo dado, de dimensiones `nrows x ncols` y devuelve un puntero a dicha imagen. Todos los píxeles se inicializan a cero. Devuelve `NULL` en caso de error.

- `int ROWS(img)`
  `IMAGE *img;`
  Dado un puntero a una imagen, devuelve su número de filas.

- `int COLS(img)`
  `IMAGE *img;`
  Dado un puntero a una imagen, devuelve su número de columnas.

- `char *NAME(img)`
  `IMAGE *img;`
  Dado un puntero a una imagen, devuelve un puntero a su nombre de archivo (sin tener en cuenta su ruta completa).

- `int img_getpixel(img, row, col)`
  `IMAGE *img;`
  `int row, col;`
  Dado un puntero a una imagen y unas coordenadas fila/columna, esta función devuelve el valor del píxel con dichas coordenadas en la imagen.

- `void img_setpixel(img, row, col, value)`
  `IMAGE *img;`
  `int row, col, value;`
  Dado un puntero a una imagen, unas coordenadas fila/columna y un valor entero `value` (en el intervalo [0, 255]), esta función fija el valor del píxel de dichas coordenadas en la imagen al valor dado.

- `int img_write(img, filename)`
  `IMAGE *img;`
  `char *filename;`
  Dado un puntero a una imagen y un nombre de archivo, guarda la imagen en el

disco con dicho nombre de archivo. Devuelve un 1 si no hay errores, 0 en caso contrario.

- `void img_free(img)`
  `IMAGE *img;`
  Dado un puntero a una imagen, libera toda la memoria asociada a la misma.

- `IMAGELIST *imgl_alloc()`
  Devuelve un puntero a una nueva estructura `IMAGELIST`, que únicamente es un vector de punteros a imágenes. Dada una `IMAGELIST *il`, `il->n` es el número de imágenes en la lista. `il->list[k]` es el puntero a la k-ésima imagen de la lista.

- `void imgl_add(il, img)`
  `IMAGELIST *il;`
  `IMAGE *img;`
  Dado un puntero a una lista de imágenes y un puntero a una imagen, añade la imagen en el último renglón de la lista.

- `void imgl_free(il)`
  `IMAGELIST *il;`
  Dado un puntero a una lista de imágenes, libera su memoria asociada.

- `void imgl_load_images_from_textfile(il, filename)`
  `IMAGELIST *il;`
  `char *filename;`
  Toma un puntero a una lista de imágenes y un nombre de archivo (un archivo que contiene una lista de rutas de imágenes, una por línea). Cada archivo de imagen en la lista se carga en memoria y se añade a la lista de imágenes `il`.

- `IMAGE *img_alloc()`
  Reserva espacio de memoria.

hidtopgm

`hidtopgm` toma el siguiente conjunto fijo de argumentos:

`hidtopgm` *net-file image-file x y n*

net-file es el fichero que contiene la red neuronal en la cual se encuentran los pesos de las neuronas ocultas que buscamos.
image-file es el fichero en el que guardaremos la imagen obtenida.
x e y son las dimensiones en píxeles de la imagen con la que se ha entrenado la red

neuronal.
n es el número de la neurona oculta objetivo. n puede variar de 1 al total de neuronas ocultas en la red.

outtopgm

`outtopgm` toma el siguiente conjunto fijo de argumentos:

`outtopgm` *net-file image-file x y n*

net-file es el fichero que contiene la red neuronal en la cual se encuentran los pesos de las neuronas ocultas que buscamos.
image-file es el fichero en el que guardaremos la imagen obtenida.
x e y son las dimensiones de las neuronas ocultas, donde x es siempre 1 + el número de neuronas ocultas especificadas por la red e y es 1.
n es el número de la neurona de salida objetivo. n puede variar de 1 al total de neuronas de salida en la red.

Apéndice C

Funciones código y OpenCV

En la fase de tratamiento de la imagen, empleamos varias funciones de OpenCV que explicaremos a continuación. Además, se exponen otras funciones desarrolladas para distintos propósitos del algoritmo.

OpenCV

- `IplImage *cvLoadImage(filename, iscolor)`
  `const char* filename;`
  `int iscolor;`
  Carga una imagen desde un archivo. El argumento `iscolor` nos especifica si la carga de la imagen va a ser en color (RGB, `CV_LOAD_IMAGE_COLOR` mediante `define` o valor positivo de `iscolor`), escala de grises (`CV_LOAD_IMAGE_GRAYSCALE` o valor 0 de `iscolor`) o sin cambios respecto al fichero de origen (`CV_LOAD_IMAGE_UNCHANGED` o valor negativo de `iscolor`). Soporta los formatos de imagen BMP, DIB, JPEG, JPG, JPE, PNG, PBM, PGM, PPM, SR, RAS, TIFF, TIF.

- `int cvSaveImage(filename, image)`
  `const char *filename;`
  `const CvArr *image;`
  . Guarda una imagen en un fichero. La imagen se guardará en el formato que indique la extensión de `filename`.

- `IplImage *cvCloneImage(image)`
  `const IplImage *image;`
  Realiza una copia completa de una imagen dada por `image`. El puntero que devuelve apunta a la imagen clonada.

- `void cvSmooth(src, dst, smoothtype, param1, param2, param3, param4)`

```
const CvArr *src;
CvArr *dst;
int smoothtype, param1, param2;
double param3, param4;
```

. Filtra la imagen `src` y guarda el resultado en `dst`. El argumento `smoothtype` nos permite elegir el tipo de filtro empleado:

- `CV_BLUR_NO_SCALE`: convolución lineal con un kernel de 1's de dimensiones `param1`×`param2`.
- `CV_BLUR`: mismo que el anterior, pero normalizando con las dimensiones de la máscara.
- `CV_GAUSSIAN`: convolución lineal con un kernel gaussiano. `param3` indica la desviación típica de la distribución.
- `CV_MEDIAN`: filtro de la mediana (kernel `param1`×`param1`).
- `CV_BILATERAL`: filtrado bilateral (usando `param3` y `param4` para definirlo).

- `void cvThreshold(src, dst, threshold, maxValue, thresholdType)`

```
const CvArr *src;
CvArr *dst;
double threshold, maxValue;
int thresholdType;
```

. Igual que en `cvSmooth`, `src` y `dst` son las imágenes fuente y destino. Esta función binariza la imagen original mediante umbralización. Este valor umbral viene dado por `threshold`. Tenemos varios tipos de umbralización disponibles según el valor del argumento `thresholdType`: `CV_THRESH_BINARY`, `CV_THRESH_BINARY_INV`, `CV_THRESH_TRUNC`, `CV_THRESH_TOZERO`, `CV_THRESH_TOZERO_INV`, `CV_THRESH_OTSU...`

- `inline CvPoint cvPoint(x, y)`

```
int x, y;
```

Función para crear una estructura CvPoint.

- `void cvRectangle(img, pt1, pt2, color, thickness, lineType, shift)`

```
CvArr *img;
CvPoint pt1, pt2;
CvScalar color;
int thickness, lineType, shift;
```

Esta función dibuja un rectángulo entre los puntos `pt1` y `pt2` de la imagen `img`.

Podemos elegir el grosor de línea y el color mediante el resto de argumentos de entrada.

- `CvScalar cvGet2D(img, i, j)`
  `const CvArr *img;`
  `int i,j;`
  Devuelve el nivel de gris de la coordenada `i,j` de la imagen `img`.

- `void cvSetImageROI(image,rect)`
  `IplImage *image;`
  `CvRect rect;`
  Define una ROI (Region of Interest) dentro de la imagen `image` para una rectángulo dado por `rect`.

- `void cvResetImageROI(image)`
  `IplImage *image;`
  Elimina la ROI de la imagen.

- `IplImage* cvCreateImage(size, depth, channels)`
  `CvSize size;`
  `int depth, channels;`
  Crea un puntero a imagen y reserva espacio de memoria. `size` indica las dimensiones de la imagen a crear, `depth` el número de bits de los elementos de la imagen y `channels` el número de componentes de cada píxel.

- `void cvResize(src, dst, interpolation)`
  `const CvArr *src;`
  `CvArr *dst; int interpolation;` Esta función redimensiona una imagen `src` a las dimensiones de `dst`. Disponemos de 4 tipos de interpolación: `CV_INTER_NN`, `CV_INTER_LINEAR`, `CV_INTER_AREA`, `CV_INTER_CUBIC`.

- `void cvReleaseImage(image)`
  `IplImage **image;`
  Libera el puntero a la imagen `image` y sus datos.

Funciones código

- `long long timeval_diff(difference, end_time, start_time)`
  `struct timeval *difference, *end_time, *start_time;`
  Función empleada para calcular intervalos de tiempo.

- `float *proyeccionh (img, iniciox, inicioy, ancho, alto)`
  `IplImage *img;`
  `int iniciox, inicioy, ancho, alto;`
  Función que calcula las proyecciones horizontales de una imagen `img`. Dada una región de dimensiones `ancho` y `alto` y un punto inicial, devuelve un puntero con las proyecciones de cada una de las filas.

- `float *proyeccionv (img, inicio, altura)`
  `IplImage *img;`
  `int inicio, altura;`
  Análogamente a la función anterior, calcula las proyecciones verticales de una región de `img` de altura determinada y lo devuelve en un puntero.

- `IplImage* normalizar(img1, rect, dim)`
  `IplImage *img1;`
  `CvRect rect;`
  `int dim;`
  Esta función define una ROI dada por `rect` en la imagen `img1`, la redimensiona a un imagen de `dim`×`dim` y devuelve dicha imagen.

Bibliografía

[1] DE LA ESCALERA HUESO, Arturo. *Visión por computador: fundamentos y métodos*. Cap. 3, 5 y 6. Prentice Hall. 2001.

[2] ISASI VIÑUELA, Pedro; GALVÁN LEÓN, Inés M. *Redes de neuronas artificiales. Un enfoque práctico*. Cap. 1,2 y 3. Pearson Prentice Hall. 2004.

[3] BORRAJO MILLÁN, Daniel; GONZÁLEZ BOTICARIO, Jesús; ISASI VIÑUELA, Pedro. *Aprendizaje automático*. Sanz y Torres. 2006.

[4] BRADSKI, Gary; KAEHLER, Adrian. *Learning OpenCV*. O'Reilly. 2008.

[5] SÁNCHEZ FERNÁNDEZ, Carlos Javier; SANDONÍS CONSUEGRA, Víctor. *Reconocimiento óptico de caracteres (OCR)*. Universidad Carlos III de Madrid.

[6] ELMORE, Megan; MARTONOSI, Margaret. *A Morphological Image Preprocessing Suite for OCR on Natural Scene Images*. Georgia Institute of Technology, Princeton University.

[7] GATOS, B.; PAPAMARKOS, N.; CHAMZAS, C. *Pattern Recognition*. Volumen 30, No. 9, pp. 1505-1519: Skew detection and text line position determination in digitized documents. Elsevier Science Ltd. 1997.

[8] CHENG, Qing. *Evaluation of OCR Algorithms for Images with Different Spatial Resolutions and Noises*. Ottawa-Carleton Institute for Electrical Engineering. 2003.

[9] WIKIBOOKS. Artificial Neural Networks. Disponible en `http://en.wikibooks.org/wiki/Artificial_Neural_Networks`. 2010.

[10] MARCOS GARCÍA, Juan José. *La escritura gótica*. Disponible en `http://guindo.pntic.mec.es/~jmag0042/paleo.php?d=escritura_gotica.pdf`. 2011.

[11] von AHN, Luis; MAURER, Benjamin; Mcmillen, Colin; ABRAHAM, David; BLUM, Manuel. *reCAPTCHA: Human-Based Character Recognition via Web Security Measures*. Disponible en `www.sciencemag.org`, Volumen 321. 12 septiembre 2008.

[12] *opencv v2.1 documentation*. Disponible en `http://opencv.willowgarage.com/documentation/c/genindex.html`. 2010.